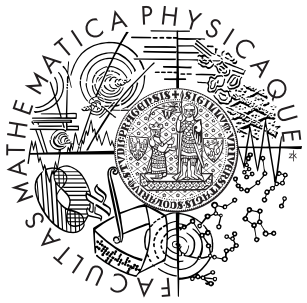
Charles University in Prague
Faculty of Mathematics and Physics

**MASTER THESIS**




Tomáš Pop

# The Progress Run-time Architecture

Department of Software Engineering

Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study Program: Informatics, Software Systems

2009

I would like to thank my mentor Jan Carlson — not only for his really important comments and priceless remarks, but also for his support with technical and administrative aspects of the thesis.

I would also like to thank Anders Pettersson for his advice on realtime systems programming issues.

I also thank Tomáš Bureš for opening a chance to me to write this thesis.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 3. 4. 2009 Tomáš Pop

# Contents

Název práce: The Progress Run-time Architecture
Autor: Tomáš Pop
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.
e-mail vedoucího: Tomas.Bures@mff.cuni.cz

Abstrakt: Práce je součástí rozsáhlejšího výzkumného záměru s názvem Progress, který usiluje o poskytnutí metod pro vývoj komponentových realtime systémů pro embedded zařízení. Jednou z nových myšlenek Progressu je sdružování komponent do větších celků nazvaných *virtuální zařízení*. Důvodem k tomuto sdružování je dosažení vyšší efektivity a také možnost vyšší míry abstrakce hardware cílových výpočetních jednotek. Tato práce začíná zkoumání implementace struktur komponentového modelu, který je součástí Progressu. Cílem práce je otevřít nutné otázky týkající se implementace vnitřní struktury virtuálních zařízení, implementace běhového prostředí a mechanismů nutných k běhu virtuálních zařízení na fyzickém zařízení. Součástí práce je i ukázková implementace běhového prostředí pokrývající lokální komunikaci a komunikaci prostřednictvím Ethernetu, implementaci podpory pro událostmi řízené a periodické úlohy a systémy s více spolupracujícími zařízeními.

Klíčová slova: Komponentové systémy, Progress, Realtime, Embedded zařízení

Title: The Progress Run-time Architecture
Author: Tomáš Pop
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Supervisor's e-mail address: Tomas.Bures@mff.cuni.cz

Abstract: This thesis is a part of a bigger research vision called Progress which aims at providing component based techniques for the development of realtime embedded systems. Progress introduces the concept of a virtual node in order to increase the effectiveness of constructed systems and improve hardware abstraction. The thesis starts research of the runtime structures of the Progress component model. The thesis aims at identifying necessary questions about the runtime internal structure of virtual nodes and about the supporting mechanisms needed to run virtual nodes on destination hardware. A part of this thesis is also a sample implementation of the virtual node runtime environment covering local and Ethernet communication, event driven and timer driven tasks, and multiple computational nodes.

Keywords: Component systems, Progress, Realtime, Embedded devices

# Chapter 1

# Introduction and Background

Models and components have become an important part in the development of embedded systems. They reduce the complexity of embedded systems and provide a formal ground on which an analysis and a synthesis may be performed. Although the component-based development for the realtime embedded systems has many significant advantages, it is still facing challenging research problems.

The programming of embedded and realtime systems differs from other programming activities. Developers can not afford memory consuming concepts like a garbage collection or complicated, but well known and tested environments like CORBA. Although the price of hardware is declining every day and embedded systems performace grows, programmers still have to be carefull about resource requirements. Another problem is strongly heterogeneous hardware and operating systems (OS) of embedded devices. This leads to conflicts between a reusability and an effective utilization of resources. The realtime approach, which is necessary for industrial applications requiring schedulability and timing analysis, brings several specifics and constraints on the programming of components themselves as well as on a runtime environment architecture.

This thesis is part of a large research vision called Progress which aims at providing component-based techniques for the development of realtime embedded systems. More specifically, the thesis should be the first step of the ProCom[1] runtime environment implementation, which is part of Progress.

The theoretical background of the thesis will be discussed in the following paragraphs, and a special emphasis will be given on the component based development and the virtual node concept, which has been introduced in the Progress research.

[1] Progress component model described in Section 1.2

## 1.1 Component Based Development

One of the biggest of today's software problems is a growing complexity. For example the complexity of the electronic devices in the vehicular domain is growing exponentially [8]. The maintainability of such systems becomes more and more problematic.

A possible solution to this problem is in dividing software into small independent units called *components*. These units have well defined interfaces — the implementation is hidden and could be possibly changed without any impact to the rest of the system. There exist more formal definitions of the component, we can cite here the Szyperski definition from [14]: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".*

Componets can be stored in a repository and systems can be built by assembling these units which were already developed, analyzed and tested. These units are reusable and can have separate life cycle. This concept potentially increases the productivity, lower the cost and improve the quality of the software systems.

Another significant advantage of the component based development is that the system could be build by assembling from prepared components. Such components could be pre-compiled and delivered by the hardware manufacturer. This approach can lower the required level of the programming skils, hide hardware details and simplify the development process.

However, this approach also has several problems. An overview could be found in [7], and here is the list of the biggest ones.

**Specification of requirements for a component** Because the component implementation is hidden and can be changed, the specification of the component should be complete and exact. The change of an internal implementation of one component should not cause requirements for change for any other component in the system.

**reusability and effective utilization of resources** To write reusable components mean in fact to write a more general code than needed for a single instance. A logical consequence is a possible suboptimal utilization of resources. This problem is on embedded systems (with specific hardware) really challenging.

**Development of components** The effort to create components and to assemble the constructed system from them for the first time is much higher than to create a usual application.

**Right level of granularity** Too small components causes a high overhead and a complicated process of system composition. Too big components do not solve the problem of complexity.

## 1.2 Progress and the ProCom Component Model

Progress is one of the research visions hosted by the Mälardalen Real-Time Research Centre. The purpose of Progress is to provide theory and tools for a cost-efficient engineering and re-engineering of component based software, mostly intentended for embedded systems. Because Progress is primarily intended to be used in the vehicular, telecommunication and automation industry, strong emphasis is given to a time analysis and reliability of modelled systems. Progress funded by from the Swedish Foundation for Strategic Research.

For illustration, in the vehicular industry it is very important to have a low production cost, because cars are produced in large quantities [4]. Today's vehicles contain many computational nodes connected with several busses. For example, Volvo CX90 has around 40 computational nodes [4]. On these nodes are running often safety-critical systems, which require a realtime approach and time analysis. An effective utilization of hardware is an important issue, because it can potentially lower the number of computation nodes. Thus the final price can be also lowered.

**Example 1.1 (Anti-lock breaking system)**
As a demonstrative example of the Progress concerns could be used the anti-lock breaking system (ABS) from the automotive domain. ABS in a car is composed of wheel sensors, brake actuators and a computation node with specific real-time software. All these parts have to communicate with each other and also communication with other car systems will be needed. Obviously, the time analysis of every single ABS part and the analysis of the communication will be required. An important question for the manufacturer of the car will be the utilization of the computational node and a possible gathering of more systems on this node.

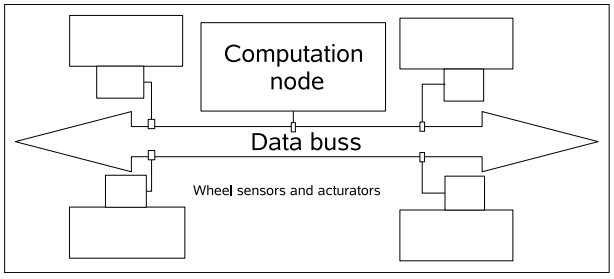



Figure 1.1: Simplified schema of ABS car subsystem

One part of Progress is the *ProCom* component model [3] which is primarily designed to be used on the embedded devices driven by realtime operating system (RTOS). ProCom and Progress introduce several new concepts. Components are

from the Progress point of view main units of the whole live cycle of the system. Also the problem with the right level of a granularity of the system is solved by different levels of decomposition of the modeled system, which is in ProCom modelled on two different levels — *ProSys* and *ProSave*.

ProSave, the lower layer, consists of small, passive components which could be hierarchically structured (atomic and composite units exist in the model). Computation on the ProSave level is based on the pipe and filter paradigm. The functionality of the ProSave component is described as a set of services. Every component can have several independent (and possibly concurrently running) services. The communication between componets is realized by *ports*. Each service contains one *input port group* and *output port groups*. An input port group is a set of input data ports and an input triggering port. Similarly, an output port group is a output triggering port and set of output data ports. These ports provide an intercomponent connection. Passivity means that components can not start any activity themselves. The services can be started only by triggering the input port.

These ProSave components are assembly units for ProSys subsystems. The ProSave level is not very important for this thesis. More information about it could be found in [3].

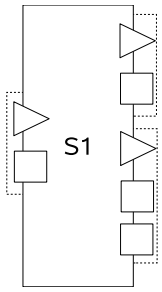



Figure 1.2: ProSave component with one service, one input group and two output groups

ProSys, the upper layer, consist of a number of concurrent subsystems. These subsystems can run potentially on several computation hardware units, called *physical nodes*. Physical nodes represent instances of real hardware. A subsystem is on this level composed of a set of concurrent functionalities that can be either *event driven* or *periodic* (driven by system timer). A consequence is that the components on this level can start a system activity. Subsystems are independent units — the only supported way, how they communicate with each other are *channels*. The channels are communication entities used for sending messages. Subsystems are connected to channels by *input and output ports*. The channels are strictly typed, only one type of message can be transmitted via one channel. Delivered messages are one type of events for an event driven functionality inside subsystems. More complex channels, with more message senders and more receivers, are also allowed, and the channel can interconnect subsys-

tems mapped to different physical nodes. Because subsystems can be reused, they can not know the structure of the channels they are using. The channel topology can not influence their characteristics.

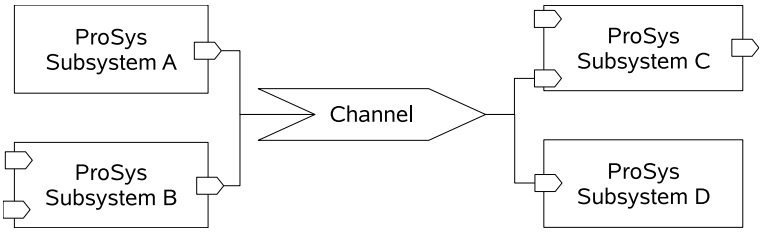



Figure 1.3: ProSys components connected with channel

It is very important for this thesis, that Progress is not intended to be used in systems with dynamic changes of the structure. After a deployment [2] is proceeded, system is never changed.

## 1.3 Software development according to Progress

The developement process in Progress starts with a set of requirements for the constructed system. Output of this process is software (typically an executable binary file or files with an appropriate configuration) for the particular target platform. The Progress development process is presented by a *software design* and a *software deployment* [10].

The software design process adds details about a system decomposition and the logical structure of the developed software. The constructed system is modelled using the ProCom design entities (ProSys subsystems and ProSave components).

The deployment process is much more important for understanding this thesis. This term means all the activities needed to transform subsystems and the system model to the executable form. Deployment includes the mapping of the design entities to virtual nodes and to a destination hardware, and the creating of executable files and their configurations.

It is advantageous to gather during the deployment process all the subsystem which are in all situations supposed to run on the same physical node to bigger units. On this account ProCom comes with a new concept of *virtual node (VN)*. As was already mentioned, programming for embedded systems is much more sensitive to overhead and efficiency issues. The VN concept gives us a great chance to optimize it. For

[2] The process of trasnforming system model and components to executable form, it is described in Section 1.3

example, the communication via a channel could be finally realized only by writing to a shared variable, if the sender and receiver subsystems belong to the same VN. This can significantly lower the overhead.

Another reason is hardware abstraction. The virtual node concept allows developers to make an abstraction layer above all the considered platforms and model the requirements for hardware in a more general way. Components are developed for virtual nodes and not for particular hardware or an OS platform. This increases possible reusability of the components and give us even a chance to prepare subsystems for platforms which are not known yet.

Now all the information needed to shortly describe the deployment process according to Progress have been mentioned. The process is shown in Figure 1.4. The order of these steps is not strictly given. It consists of the following parts [10]:

- Allocation of subsystems (components) to virtual nodes.
- Synthesis of the code belonging to each virtual node.
- Mapping virtual nodes to physical nodes.
- Creating of the glue and system wrapper code for previously synthesized virtual node code and building final executable binaries.

In the virtual node synthesis, the code of subsystems are merged, optimized and mapped to the primitives of the underlying operating system. In the creating of the glue and system wrapper code, the communication between tasks from different virtual nodes is solved and a the main executable file, responsible for running appropriate virtual nodes for each physical node, is created.

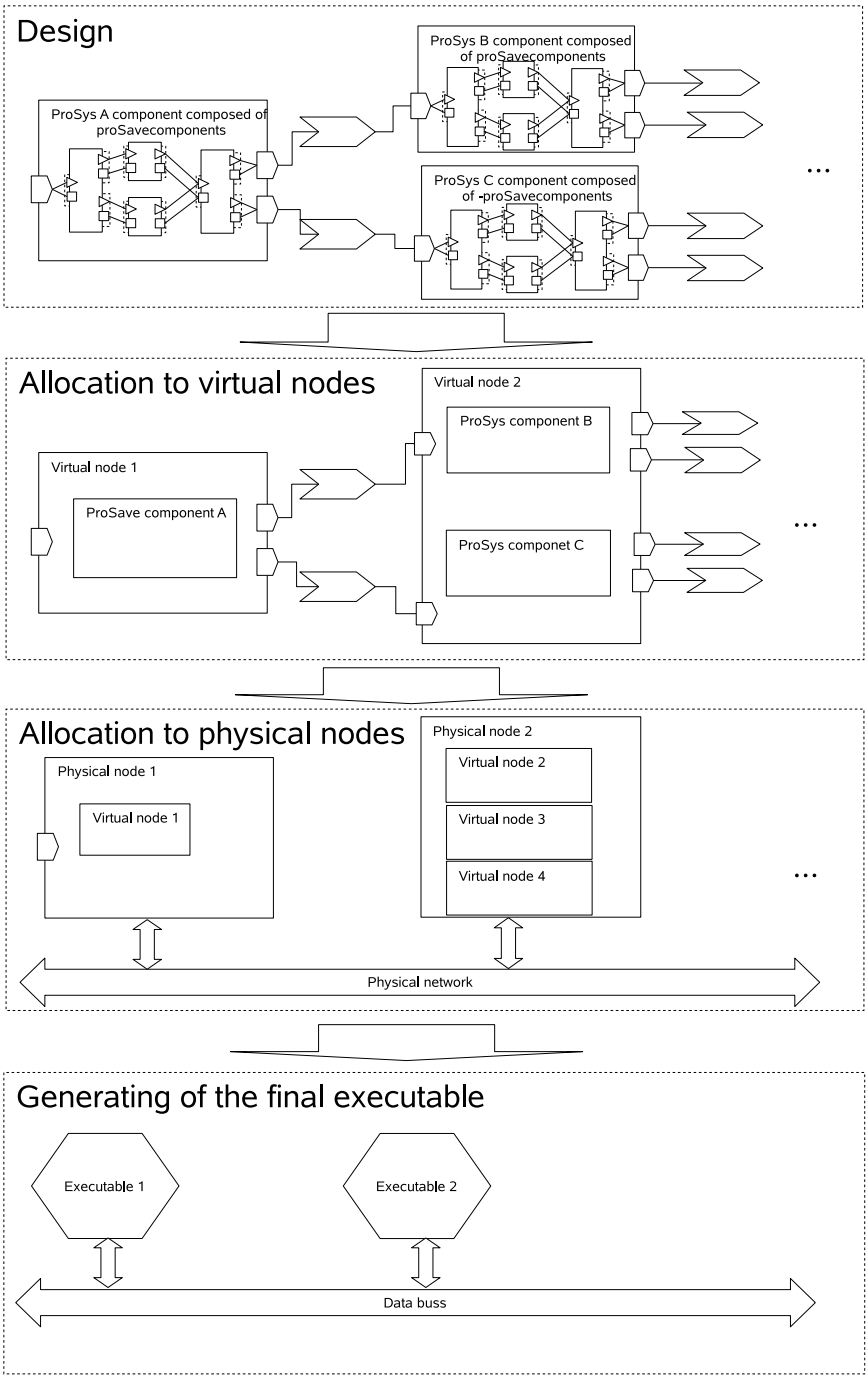



Figure 1.4: Software deployment according to PROGRESS

# Chapter 2

# About This Thesis

Progress and ProCom are in a research and development stage. A lot of theoretical work has been done and now is the right moment to support this work with the first prototype implementation. This implementation tests existing concepts and theoretical ideas in a practice, opens new questions, and discuss possible ways, how some issues can be implemented.

This prototype and this thesis can be a starting point for the future implementations of the ProCom runtime environments.

## 2.1 Thesis Purpose

The thesis should investigate the runtime structure of the virtual nodes and the supporting mechanisms needed to run them on physical hardware, including the support for communication between virtual nodes. The information about the internal structure and the design of a virtual node are necessary for the tools intended to be used for the VN code synthesis. Better information about the structure of the generated code and a pre-compiled parts of the runtime environment can help in the development of the Progress IDE.

There are many unresolved issues connected to the ProCom runtime structures and this thesis does not aim at providing the final implementation of the ProCom runtime environment. It is just the start of research of these structures and the implementation provided as a part of the thesis was developed only to prove the functionality of presented ideas and to identify hidden problems.

*The biggest reason why the work on this thesis was done is not to find the solutions, but to identify and open the right questions.*

This fact and a limited time are the reasons why some parts of the runtime are slightly simplified.

General ideas, examples and details of sample implementation are separated as much

as possible in the text, even if it is often problematic to say exactly what is still in the general part and what is already part of the implementation.

## 2.2 Thesis Structure

This thesis is composed of two parts. The first is the report, where the identified issues are discussed. The second is a sample prototype of the ProCom model runtime environment and a simple component based application running in it. The implementation part, which was written first, helps with identifying concerns and demonstrating possible solutions. The prototype covers handling of a local and a remote communication, event driven and time driven actions and systems with more than one physical node.

The application is introduced first, the discussion part follows. At the end of the thesis the important decision of the implementation are discussed in detail.

## 2.3 Limitations and Scope of the Thesis

Progress is a huge vision and this work has a limited scope. The sample runtime environment does not aim to be part of the final deployment tools. Even if the implementation is trying to be general as possible, it is still restricted only to the chosen hardware platforms, operating systems and message transmissions media and also some of the possible design features were omitted. For example, virtual nodes are considered as indivisible units. It is not yet know, if this assumption will be fullfilled in the final implementation because generation of code from the ProSave components and ports and the virtual node interface are not addressed yet. Also the exact semantic of channel operations is not yet known.

Many other aspects have not been elaborated yet and deeper research and discussion would be necessary to find the right answer. The thesis is not trying in mentioned situations to find the best or the final solution. Instead, the issue is usually solved in the simplest way which is not conflicting with the already existing research, and the vagueness is documented.

## 2.4 Sample Application

Figure 2.1 shows the schema of the sample application. This system consists of three virtual nodes. These virtual nodes will in the future be synthesized of subsystems by a code synthesizer which will be a part of the Progress IDE. The system is finally deployed for two configuration — mapped on one and on two physical nodes. It demonstrates the possibility of mapping the same virtual nodes to physical nodes in several different ways. The application is written in a “task style”. There is a set of concurrent

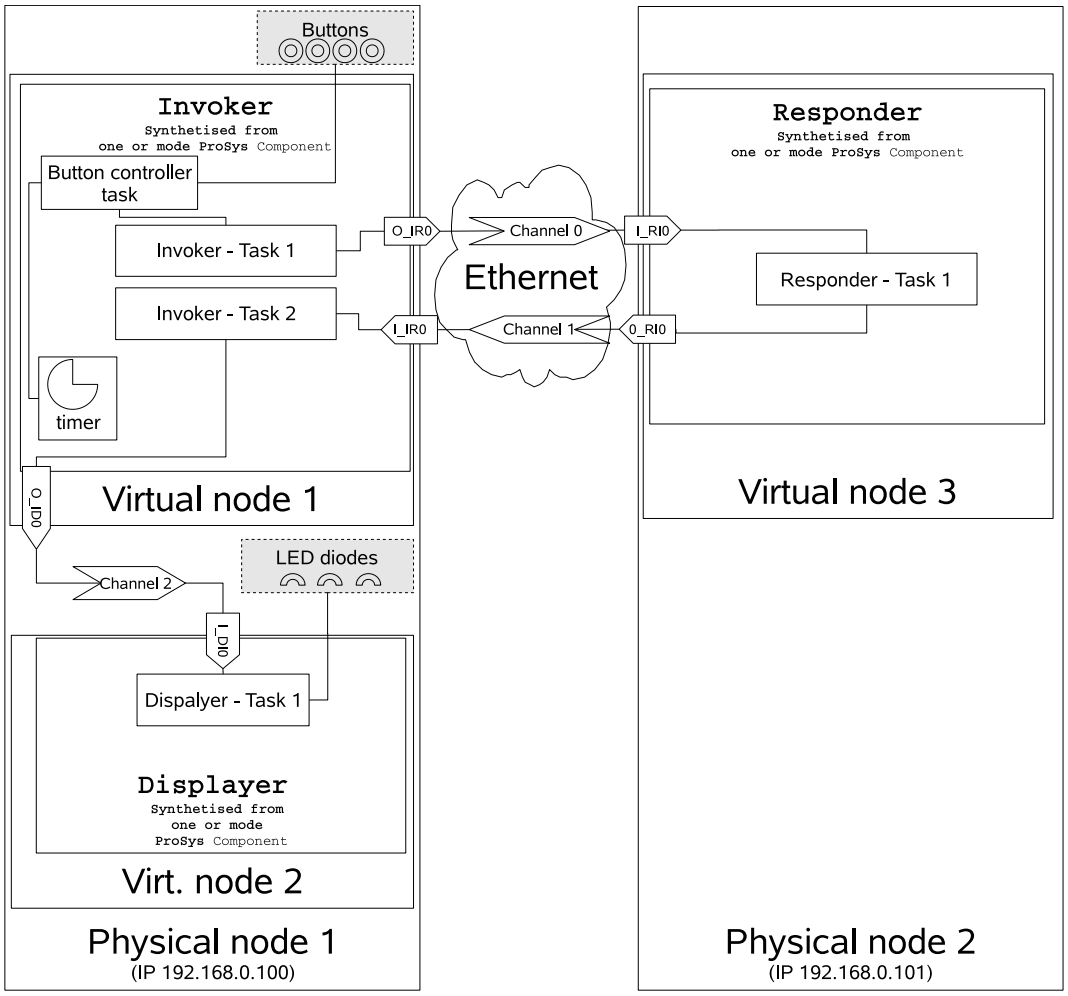



Figure 2.1: Sample application

tasks running on each physical node. The final application is simple, but it shows used concepts of the implemented runtime environment.

The sample application is finally running on the embedded Arcom VIPER board. This board has four switches and three led diodes. This hardware realizes sensors and and visualizes outputs. Arcom Embedded Linux operating system and eCos RTOS has been chosen as the example of operating systems. Experiments with a runtime environment ported on both systems has the following reason: eCos and Linux are very different systems and the implementation on both give us a wider view on the problem.

- eCos is a low-level system designed for embedded devices, it does not separate the user space and the kernel space. It is a satisfactory example of embedded RTOS. Unluckily, the support of the Ethernet controller on the board is not working properly.
- Linux is a robust system with separated kernel space and user space. There is no way to control hardware on the board or interrupts directly (from a userspace). The network support is working and we can demonstrate the distributed communication. Linux which is running on the board does not contain a realtime scheduler (it is not realtime Linux)

In the Linux implementation the POSIX API is used, in the eCos native kernel API.

The basic functionality of the application is very simple. If the user presses the first button, the green led diode is lighted up. Any other button means a "bad answer" and pressing it switch on the red led diode. Also some debug messages are printed on standart output to show what is happening inside.

Names are given to the virtual nodes used in the application. The first reason to do it is much simpler writing about them but these unique identifiers have also a technical reason, which will be discussed later. The main example application will be composed of three channels and the following virtual nodes:

**Invoker** This subsystem has two tasks to simulate the situation, when a virtual node was synthesized from more than one subsystem. The first task of Invoker reads the data generated by the sensor task. Every time data come, it will periodically send a message using the output port *O_IR0*. The second Invoker task is in the role of the forwarder. It is waiting for a message from the *I_IR0* port. When the message arrive, it is forwarderd to the port *O_ID0*.

**Responder** subsystem receives the data from *I_RI0* port and generate the answer. In this case, the Responder subsystem does not perform any operation and simply replies read char using the *O_RI0* port.

**Displayer** subsystem is displaying messages received from the *I_DI0* port. In the final implementation on the Viper board this means, that the Displayer is switching on and off the led diodes on the IO panel of the board.

## 2.5 Related Works

The Pecos project [12] was probably most similar to this work, one of it's major achievement was to create prototype of runtime environment for component based realtime systems running on embedded devices. This project is no more running, it's web pages are not accessible and it was hard to find detail information about this research.

A lot of runtime environments for different component systems exists. As example could be mentioned SOFA [9] or CCM implementation, but these environments are not suitable for the Progress research vision, because they are not intended to be used on embedded devices, they are often implemented in Java and are too general (too resource consuming) for Progress.

The environment considered in this thesis can take full advantage of ProCom specifics described in Section 1.2, for example of no dynamic changes of the system structure in runtime.

Many important information and ideas could be also found in other existing component runtime environments like COM or JavaBeans, but this projects were originally designed for other application domain, are very complex and hard to understand.

Another great source of inspiration was the SaveOS project [11], which is also hosted by the Mälardalen Real-Time Research Centre. Its purpose was to create library providing operating system abstraction for realtime applications created using saveCCM language [1]. This project is similar only to a part of the problems discussed in this thesis - it does not care about the component communication and about their access to a device specific hardware. Importance of inspiration coming from this project is an appropriate RTOS primitives abstraction API design.

# Chapter 3

# The Progress Runtime Environment Concept

The term "Runtime Environment" is used in this thesis to describe all the supporting mechanisms needed to run virtual nodes on destination hardware. The design of such mechanism and the structure of the VN are very closely interconnected issues. In this chapter the general requirements on the runtime environment are described as well as some hi-level decisions made in the early phase of the runtime implementation.

## 3.1 Requirements

There is a set of requirements on the runtime environment and the virtual node structure coming from the semantics of the ProCom model described in [3] and from the logic of the Progress deployment process described in [10]. Some of these requirements are explicitly specified in the mentioned texts, others are logical consequence of these documents.

The most important requirements are:

- A virtual node should be indifferent to the rest of the environment.
- It must be strictly separated, what modelling entity is responsible foe each individual part of the final source code.
- The runtime should provide the way of virtual node communication (channels).
- The channel topology can not influence behavior and characteristic of the VN performing channel operations.
- At least one type of semantics supported by the channel mechanism is clearly defined and have to be implemented.

In addition to these requirements there exists a set of demands, which do not have to be valid under all circumstances, but they can simplify the development of other parts of the Progress IDE and it is advantageous to take them in an account.

- The structure of the virtual node should be easy to generate from components.
- The code needed to run virtual nodes consists of two parts: generated source code, which will be produced for each deployment process, and a pre-compiled "runtime library". The amount of the generated code should be as small as possible and it should be as easy-to-generate as possible.
- The structure of a virtual node should be as simple as possible. The runtime environment is programmed only once and could be complicated, because there will be time to test it.
- The runtime environment should provide access to all (even possibly) shared resources. This can avoid conflicts in resource access, increase the reusability and simplify the code generation. This requirement is not so easy to fullfill because of the conflict between a reusability and an effective utilization of resources.
- It should be simple to assemble executable from the prepared virtual nodes.

As was already written in Section 1.2, no dynamic changes are required after the deployment process is finished. This means, that there is not necessary to implement any mechanism to change a channel topology at runtime or to add or remove virtual nodes. All the information about channel structure and included components can be included in the generated C code.

It is advantageous, to fulfill all these requirements and facts, to divide runtime environment architecture into three interconnected parts.

**Abstraction layer** provides access to shared resources and the underlying system.

**Channels** cover the inter-virtual-node communication.

**Support for physical and virtual nodes** enables simple structure of a VN and easy assembling of final executable.

## 3.2 Abstraction Layer

An abstraction layer helps developers to produce reusable code and simplifies the code synthesis. If the level of generality is right, the code can be successfully used on several operating systems and hardware platforms.

The abstraction layer is composed of two parts: the operating system abstraction and the hardware abstraction. The first part provides abstraction of the system primitives like semaphores, mutexes, tasks or periodic actions. The same primitives are common in the major part of existing realtime systems. If the architects of the runtime environment decide to use some more specific artifacts (like for example conditional variables), the runtime will be probably not portable to some operations systems.

The second part is the hardware abstraction. The problem is not in the low-level hardware abstraction, which is usually done by the operating system, but in the abstraction of access to device specific hardware and in its access management. Because of the reusability of the code, VNs will not typically use a direct access to the real hardware of an embedded device but it will use the hardware abstraction provided by the runtime environment. A high level of abstraction can increase reusability of the code, but it can potentially lower an effective utilization of the device hardware. This problem is ilustrated in Figure 3.1 and in Example 6.

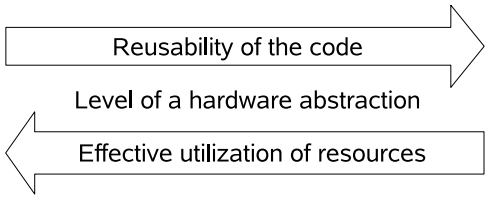



Figure 3.1: Problem of the right level of abstraction

## 3.3 Channel Support

Channels are the only supported way in which virtual nodes can communicate with each other (a deeper discussion of this topic is given in Chapter 6). The "channel" is also an abstraction, systems can include more types of a transmission media, but the access to channels is uniform. Also the exact channel topology (for example the number of readers and writers or the fact whether the channel is local on a physical node or connects more hardware units) is hidden to the virtual node using it, and can not change its characteristics. This means, that the *send* and the *receive* operation take (from the virtual node point of view) a constant ammount of resources.

More about channels could be find in Chapter 6.

## 3.4 Support for Physical and Virtual Nodes

By the support for virtual and physical node is meant the code where all the structures needed to define physical and virtual node are defined, and functions responsible for

running the tasks from prepared virtual nodes, which are a part of physical nodes. The virtual node support solves the issues of a task definition, alarms and data structures needed for creating virtual nodes independently on the rest of the system. The physical node support is responsible for hardware and OS initialization, for running the virtual nodes and for the final clean-up if needed.

# Chapter 4

# Virtual Node Support

The virtual node as the ProCom and Progress concept was already described in Section 1.3. In this chapter will be discussed its possible internal structure and the way the virtual nodes are interacting with each other and accessing the resources. Different forms of an internal structure were tested during the work on the thesis. Advantages and disadvantages of these forms are discussed in the following sections.

The text is also focused on the identified implementation problems.

## 4.1 Structure of the Virtual Node

A virtual node consists of the ProSys subsystems. These subsystems are merged together during the synthesis process. Remember that the subsystem is a set of (periodic and event driven) functionalities and appropriate data structures (Section 1.2). Because the tools for subsystem synthesis have not been finished yet, the structure of a virtual node have to be determined. The virtual node merge and gather subsystems, therefore the internal structure can be also described as set of functionalities and appropriate data structures. The mentioned functionalities are in realtime systems realized by the *tasks*.

All these facts lead to the following general idea: Virtual node is logically structured as a set of data structures, periodic tasks and event driven tasks.

There are two groups of data structures inside the VN realization: data structures needed by the VN support and the structures belonging to a functionality of the virtual node tasks. The periodic tasks are timer-controlled and the event driven tasks are started by special occasions like a receiving of a message or by a change of some synchronization primitive caused by another activity in a system.

Data structures could be naturally represented in the C language, the only problem is memory allocation, because functions like *malloc()* are in many RTOS not allowed since a scheduler is started, and the data structures needed for the virtual node support

and the structures shared by multiple tasks of a single VN can not typically be declared inside any function of the VN code.

This constraint could be bypassed by using statically allocated memory. This method is very advantageous for ProCom, because no dynamic changes in the system structure are needed. However, this approach has a fundamental problem in case of using more instances of the same virtual node: the memory (as well as other resources) is in such situation shared among all the instances and collisions are possible. Issues related to the multiple instances question are discussed in Section 4.3.

In special cases, when the usage of statically allocated memory is not possible or is too complicated, a memory pool could be created in the initial part of the system execution (before the scheduler is started), and runtime mechanisms could be used to distribute the memory among the virtual nodes. Such mechanisms were not needed in the sample application, and were not implemented.

A third way is to allocate memory statically before the virtual node tasks are started and pass a pointer to this memory to the virtual node as a parameter. (More about the argument passing is in Section 4.2). This approach could be used to avoid problems with dynamic allocation, but it breaks partially the requirement of separation of VN and the rest of the system.

Tasks could be represented as C functions of a given type. The type of such function allows to pass data inside the task. Such passing is possible in some way in the major part of realtime systems.

During the experiments with the system implementation, two possible ways how the VN structure could be represented were tested in detail — the version structured as an array of tasks and the main function structured one. The first one is more general and is very close to the logical structure of the virtual node. The second one can be used only on some realtime systems (Section 2), but it would be much more comfortable to the architects of the subsystem synthesis tools.

**Example 4.1 (Structured as an array of tasks)**
This implementation very straightforwardly follows the ideas mentioned above. Event driven tasks are represented as functions, timer driven tasks are represented as periodic actions (let's look at it as the handlers of an interrupt, more can be found in Section 7.4). Data structures are allocated in statical variables in an extra part of the *.c* file, where the virtual node is stored and pointers to this variables are saved in the VN runtime members.

The start-up procedure is shown in Listing 4.1. All the virtual nodes mapped to actual physical node are added during the execution of the function *phnode_init()*. This function calls initialzation routines of each virtual node, an example of such initialization is in Listing 4.2. All the tasks are activated during the execution of the function *phnode_run()*. The *main()* function is the same for all the physical nodes, and is a part of the system library.

```
int main () {
        /* Here all VN are added */
        phnode_def = phnode_init();

        /* Here all the tasks are activated */
        phnode_run( phnode_def );
        printf ("Start process OK..\n"); fflush(stdout);
        phnode_wait_all( phnode_def );

        phnode_destroy();
        return 0;
}
```

Listing 4.1: VN structure – structured as an array of tasks – system start

```
void vn_writer_init(vnode_definition_t * vnode_def,
                                task_definition_t * tasks ){
        vnode_def->nr_task = NR_TASKS;
        vnode_def->tasks = tasks;
        vnode_def->vnode_name = "WRITER";
        vnode_def->per_actions = NULL;
        vnode_def->nr_per_actions = 0;

        tasks[0].task_fun =  vnode_reader_task_1;
        tasks[0].task_name = "WRITER, task 1";
        tasks[0].stack_ptr = task_stacks[0];
        tasks[0].stack_size = TYPICAL_STACK_SIZE;
        tasks[0].priority = task_get_min_priority();
}
```

Listing 4.2: VN structure – structured as an array of tasks

**Example 4.2 ("Main Function" structured)**
In this implementation, the start-up routine calls directly functions of the virtual nodes responsible for the virtual node starting.

```
int main () {
        phnode_init();

        vnode_start ( vnode_writer_run );
        vnode_start ( vnode_reader_run );

        printf ("Start process OK..\n"); fflush(stdout);
        vnodes_wait_all();

        phnode_destroy();
        return 0;
```

```
}
```

Listing 4.3: VN structure – "Main Function" Structured – system start (simplified)

The code of the virtual node in Listing 4.4 consists of one task (represented by function *vnode_writer_task_1*). The *vnode_writer_run()* is the "main" (wrapper) function of the virtual node. All the data structures of this VN could be allocated inside this main routine. The wrapper function runs all the tasks and returns in the moment when all the tasks defined in this virtual node finish their work.

```
#define WRITER_NR_TASKS 1

/* Runs all the virtual node tasks. */
void * vnode_writer_run(void * unused){
    int i;
    task_handle_t tasks[WRITER_NR_TASKS];
    tasks[0] = task_run( vnode_reader_task_1 );

    for (i = 0; i < WRITER_NR_TASKS; i++){
            task_wait(tasks[i]);
    }
    return NULL;
}
```

Listing 4.4: VN structure – "Main Function" Structured

The shown structure of the virtual node would be very comfortable for the virtual node synthesis process. Also there are no global variables needed for memory allocation.

Let's note that this solution needs one task to be created from another and this is not supported in all the RT operating systems — that's why this structure could be implemented only on some platforms.

## 4.2 Passing Data into the Virtual Node

In the preceding paragraphs an extra attention was paid to the fact that tasks are represented by functions, which allows data to be passed into the task. Let's have a look at which data should be available inside the virtual node.

There is at least one reasonable argument needed for VN realization: references to its input and output ports. If only one instance of each virtual node is assumed on the physical node, the outputs and inputs are the only necessary arguments. If more instances are allowed, these instances can obtain extra data to change their behavior.

We can never be sure that all potentially supported underlying operating systems will support explicit passing of data to the tasks, even if the major part do it in some way. In the other cases we can use global variables (for example hidden by some macro), so the variable looks same in all VNs, but the meaning can differ. Disadvantages of this

"global variable approach" are the complications in case of pre-compiled virtual nodes (see Section 4.7).

**Example 4.3 (Channel reading and writing - channel reference passed by a global variable)**
The channels have not been discussed in detail yet, but an understand of following listings is not difficult. In the first one, the task function type doesn't need any arguments — all the data (reference to the inputs and outputs) were passed as a global variable. WRITER_OUTPUT_0 is macro defining reference to appropriate channel port. This macro is defined in system generated code (created during the deployment process).

```
/* One task of virtual node */
task_ret_type_t vnode_writer_task_1(){
    char c;
    while (1){
        c = sensor();
        port_write(WRITER_OUTPUT_0, (void *) &c);
        task_wait_next_period();
    }
    task_ret_statement;
}
```

Listing 4.5: Writing to the channel — channel reference was passed by a global variable

For comparison, here is the fragment of code with similar function, where passing by an argument is used:

**Example 4.4 (Channel Reading and Writing - channel reference passed by an argument
)**

```
task_ret_type_t vnode_writer_task_1(task_data_t *arguments){
    char c;
    while (1){
        c = sensor();
        port_write(arguments->VN_output_ports[0], (void *) &c);
        task_wait_next_period();
    }
    task_ret_statement;
}
```

Listing 4.6: Writing to the channel — channel reference was passed by an argument

Even if the second listing looks much better, there exist situations, where the global variable approach can be advantageous. At first, global variable can be used in any system. Another reason is, that the macro (global variable) is available anywhere in

the code. In situations, when a part of the task code should be given to a subroutine, and the subroutine can not be defined inside the original function, this approach is advantageous — no subroutine arguments are needed.

Howewer, this disadvantage of the approach with passing ports as an argument could be bypassed by another features of a runtime environment, see Section 7.2.

## 4.3 Multiple instances of a virtual node

There are situations when using more instances of a single VN on one physical node makes sense. In the example of the ABS car system from Section 1, an instance of a VN could theoretically be used for an every single brake acturator.

Be very careful if more instances of a virtual node is allowed. If the memory is allocated in the static variables inside VN code, this memory space will probably be shared among all the instances of the virtual node. Also all other resources (expetially hardware entities) would be potentially accessed multiple times.

Simple duplication of the code of the virtual nodes solves only some of these issues (the statically allocated memory, not the hardware access) and brings several minor technical complications (like VN functions renaming to be able to link all the instances together).

A possible solution to this memory conflicts is to pass a pointer to non-shared memory as an argument of the virtual node realization already discussed in Section 4.1

## 4.4 Accessing Information about the Virtual Node

There are some information about the virtual node, which may be useful to the runtime environment for several reasons. For example, the runtime environment may store handles to the tasks running in it, and it need to know how much memory to allocate for these handles.

This information could be exported explicitly (for example in a header file) or they could be available by calling the query functions defined in the virtual node code or by calling query functions on the structures defined in the virtual node code.

Note, there is huge difference between information needed about VN for runtime environment and data needed for the deployment process. Probably we want to know how many ports the virtual node is using, which resources are used by a VN or other information, but the process of subsystem synthesis must take care about this issues, not the runtime environment.

Anyway, there should be defined how the data needed by the runtime environment could be accessed.

## 4.5 Virtual Node Environment

A virtual node is possibly interacting with its environment — the hardware of the device and potentially the data structures of the physical node. This section describes identified problems in the initialization of the environment and in the access to the resources.

### 4.5.1 Hardware Access

Initialization of the environment of the VN has two principal parts:

**Software initialization** where variables are set to initial values, or the memory will be allocated if system allows it.

**Hardware initialization** where for example initial values are written to the hardware registers.

The software initialization of non-shared memory could be done independently of other virtual nodes.

The hardware initialization and a possible initialization of any shared structures are more problematic. Let's image a situation when one VN wants to write 0 to some register as an initial value and another one 1. In such situations the system behavior would be dependent on an initialization routines calls ordering. This is a sign of a bad system design. The system looses the advantages of the componet approach and it is analysis is significantly more complicated. Eventhough, there is no explicit protection against this possibility.

There are different ways how to avoid similar situations. First, hardware access could be divided to several phases — low level initialization, and higher level initialization. In the low level initialization, for example, processor registers are set. This system wide options should not be accessed from the virtual nodes (this values are attributes of the destination platform). The upper level initialization can set up some hardware resource needed by virtual node using it. The access to this resource is granted only to this VN (and the deployment process take care, that no other VN will touch it) or it could be shared by more virtual nodes. In such situations the behavior of the VN should not be dependent on the initial hardware state.

The low level initialization can be implemented in the abstraction layer or in the library enabling hardware access.

The Discussion of hardware access opens the question what exactly is the responsibility of the runtime environment. Should the environment solve possible collisions in the hardware access, or is this issue the responsibility of the deployment process and at the time of execution no such collisions can occur?

Another issue is the realization of the hardware access in the code. Each subsystem is written for (or let's say could be deployed on) some virtual node. There is a set of

requirements associated with the subsystem defining propertys of the VN, which the subsystem can be mapped to, describing which primitives, functions and resources have to be available on this virtual node.

Another set of requiremts is specified for a VN. These requirements define which physical node the VN could be deployed on.

These ideas could be realized for example in the following way: Each requirement of the subsystem enables inclusion of the appropriate header file. This enable the compilation of the virtual node. If the declarations from the mentioned header files are not visible, errors occur at compile time. Deployment on physical hardware could be realized by linking appropriate libraries for the destination hardware. In this library, the functions declared in the discussed header files are implemented.

Let's explain this on following example:

**Example 4.5 (Requirement realization)**
A component is written for a VN, satisfying requirement "Storing to persistent files". This will enable subsystem realisations to call filesystem functions like *file-write()*, *file-read()*, because these functions will be declared in an included header file. These functions would be implemented in library *progress-architecture-XY.la*, specific for the destination hardware and OS platform. A similar technique was used in the sample application.

There are also other "global" resources than hardware and time, which are shared among virtual nodes, and the code synthesis tool has to take care about them. For example, if the node is using POSIX, the POSIX signals are also shared resources. If two VNs use the same signal, the result can be unpredictable. The situation is different from non-component programming. If you are using POSIX threads on Linux, you probably know that your application usually can not use SIGUSR1 and SIGUSR2, because these two can be used by the POSIX thread library. But this is another situation — you never know, which VN will be running with your VN and which signals or other resources will be used. Already developed and well tested VNs can fail! There are several possible ways how to avoid similar problems. The simplest approach is not to use these primitives at all. Othervise, methods of the abstraction layer can be used. If these resources are used directly, the mapping process should at least perform conflict checking.

### 4.5.2 Level of the Hardware Abstraction

As was already mentioned in Section 3.2, having the right level of an abstraction of the hardware access is very important. Increasing the abstraction level is turning up the reusability, but on the other hand it lowers the effective utilization of resources. Let's explain this term a little bit more on an example:

**Example 4.6 (Level of the Hardware Abstraction)**
Imagine the situation of creating a VN, which will be able to signal three possible states:

- No answer given (undefined).
- The answer is OK.
- The answer is FAIL.

This VN is primarily designed for hardware with three led diodes. But in the future it will may mapped it on another hardware. So what should the abstraction look like? Here are some possibilities:

- signal_status(status_t status); status ∈ STATUS_OK, STATUS_FAIL and STATUS_UNKNOWN
- led_set(led_t which_led, command_t how); which_led ∈ LED_GREEN, LED_RED, LED_YELLOW; how ∈ SET_ON, SET_OFF
- register_write(reg_t register, value_t value);

The first possibility maximizes the reusability of the code, the last one the utilization of possibly different hardware.

## 4.6 End of the Execution — Releasing Resources

An embedded device usually do its job until the end of execution of the system (until the system shut-down). In such situations no special clean-up is needed, because next time the whole system is started from the very beginning.

The question is, whether the resources (for example an allocated memory if any, open sockets and file descriptors, etc) should be released correctly. In a small system like eCos it might not be necessary. On the other hand, on system like Linux it makes perfect sense, because this systems are not necessarily restarted after each system execution cycle.

In some systems, thread APIs do not support mechanism similar to the POSIX function *thread_join()*, which is needed to determine the time to do the clean-up. This could be worked around for example by a posting a semaphore anytime, when a task ends its execution (which allows to implement a function like *wait_all_tasks()* ).

## 4.7 Could Virtual Node be Pre-compiled?

Because *"the trend is to deliver components in binary form"* [7], a very important question is also whether the virtual nodes could be delivered as pre-compiled units.

This issue is closely connected to the question of passing arguments and the virtual node structure. If the ports and other data are passed as arguments, the virtual node can be pre-compiled and linked to the final binary during the build process easily.

If the arguments are passed as a macro (in fact a global variable), the situation is a little bit more complicated. It would be possible to find a solution, where the macro can define global variable, which will be set at runtime, but the variable has to be reserved for this VN and the VN would not be separated from the rest of the system.

In addition, delivering of virtual nodes as precompiled units has several other disadvantages. One of them is restraint of duplication of VN instances by copying (and modifying) of their source code mentioned in Section 4.3. Considering this, the precompilation of virtual nodes bring additional complexity and complications to the build process.

# Chapter 5

# Physical Node Structure, Building the Executable Files

In this chapter, the physical node structure and the process of building executable files from virtual nodes is discussed. Because the final binaries will contain pre-compiled parts and parts generated and compiled for every individual deployment process, the last section of this chapter is focused on this issue.

## 5.1 Structure of the Physical Node

The physical node structure is primarily a container of virtual nodes, but it is reasonable if it holds also information about other entities, like *system tasks*. On the physical node can potentially run tasks not belonging to any virtual node. An example of such task is a periodic hardware controller or a *channel listener task* mentioned in Section 5.2.

In conclusion, physical node support gathers the following information:

- Virtual node definitions.
- System tasks which are not part of virtual nodes.
- Channels.
- Other (implementation dependent) information.

Two versions of a possible implementation were tested. Similarly to the structure of the virtual node, the main-function structured and the version based on a structure describing the physical node were tested. Also pros and cons of these strategies are analogous to the already discussed virtual node structures.

Listing 5.1 shows the main-function structured version of a physical node with three virtual nodes.

**Example 5.1 (Physical node structure – "main function" structured version)**

```
int main (){
    vnode_handle_t vnode_handles[NR_VNODES];
    int i;

    phnode_init();
    vnode_handles[0] = vn_start( vn_reader_run );
    vnode_handles[1] = vn_start( vn_writer1_run );
    vnode_handles[2] = vn_start( vn_writer2_run );
    execution_start( vnode_handles );
    printf ("Start process OK..\n");

    wait_execution_end( vnode_handles );

    phnode_destroy();
    return 0;
}
```

Listing 5.1: Physical node structure – "main function" structured version of a physical node with three virtual nodes

The *main()* function code is generated by the deployment process tools as well as the function *phnode_init()* and *phnode_destroy()*, where the composition and destruction of the system is described.

If the function *vn_start()* activates tasks, the already mentioned problem at running one task from another occurs. If this function return the definition of the virtual node, the solution is very similar to the definition structure based one. On the other hand, this solution gives a pretty nice freedom to the developers of the deployment tools — they can call directly all the necessary subroutines from the main function and organize the code as they need. The biggest disadvantage of this approach is an unclear separation of the physical node description from the start-up procedure. Note, that this code is generated for each individual deployment process and if the start-up process differ on different platforms the creating of this functions would possibly be complicated. This is the reason, why the solution based on a definition structure is finally used in the sample implementation.

Listing 5.2 shows the structure used to describe a physical node.

**Example 5.2 (Physical node structure – structured describing physical node)**

```
/** Structure defining physical node. */
typedef struct __phnode_definition {
        /** Number of virtual nodes */
        int nr_vnodes;
        /** Number of tasks defined in all virtual
```

```
        nodea included in this physical node. */
        int nr_tasks;
        /** Included virtual nodes definitions. */
        struct __vnode_definition *vnode_defs;
        /** Included channels definitions. */
        struct __channel  *channels;
        /** Number of the channels included. */
        int nr_channels;
        /** Number of system tasks */
        int nr_sys_tasks;
        /** System tasks definitions. */
        struct __task_definition *sys_tasks;
        /*
        ... Other implementation dependent data structures
        */
} phnode_definition_t;
```

Listing 5.2: Physical node structure – structured describing physical node

In this version, the main function is the same for all physical nodes and it is a part of the precompiled library. This *main()* function was already presented in Listing 4.1. The way to add virtual nodes to physical node could be found in appendixes.

The system composition is defined in the generated function *phnode_init()*. In this function all the definitions of the virtual nodes are added, channels are connected and data structures are initialized. After this, the *phnode_run()* function is called with the parameter of the complete physical node definition structure. This function is a part of precompiled library and solves the start-up process details for all the supported platforms.

It is necessary to be very careful about global variables and use strictly the *static* keyword. The situation is even worse than in non component-based programming. The code of virtual nodes has always the same structure and it is highly probable, that the variable names will occur repeatedly.

## 5.2 System Tasks

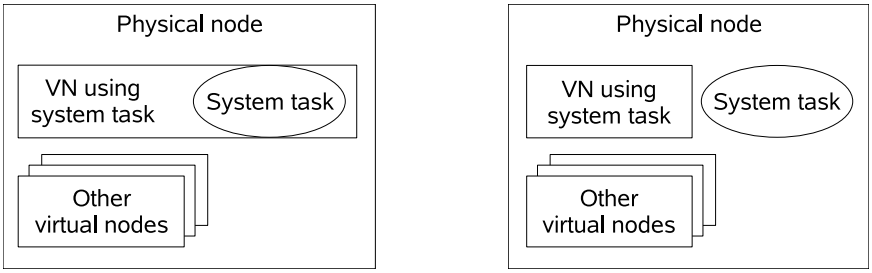
On the physical node could be defined tasks, which are necessary for system execution but which are not part of any virtual node. The question where to allocate memory for these tasks is little tricky. Here are two examples of such tasks.

**Example 5.3 (System tasks – Ethernet listener task)**
For the Ethernet channel implementation in the sample environment an extra task (*listener task*) is required for every channel and every physical node. This task is not a logical part of any virtual node, because the virtual node doesn't know the topology of

the channels and can not know about any Ethernet listener tasks. On the other hand, this listener task is not logically contained in the physical node outside the virtual node. If we map all the virtual nodes to a single physical node, there will be neither the Ethernet channel nor the Ethernet listener task.

**Example 5.4 (System tasks – button controller task)**
Second example: In the sample application, the buttons on the board are controlled by a special task, which periodically reads the button status and searches for edges in the button signal. This task is not a part of the virtual node, it doesn't do the job of the virtual node — it controls hardware. On the other hand, if this virtual node is not mapped to this physical node, the controller task will not be needed on this physical node, but somewhere else. In the sample implementation this task was placed logically to the virtual node, which uses a result of it's work.

A question is, where to place such system tasks. This decision has to be made, because it is necessary to allocate the memory for the task data structures somewhere. The situation is depicted in Figure 5.1.




(a) System tasks outside VN (b) System tasks as a part of VN

Figure 5.1: Two possible ways how to treat the system tasks

## 5.3 Generated and Pre-compiled Code

It was already mentioned in Section 3.1 that ProCom does not allow any dynamic changes in the system structure at runtime. It is possible to generate the C code during every deployment process, where the system configuration and the system structure are defined. On the other hand, the runtime environment support should be pre-compiled as much as possible.

From the paragraph above we get the following requirements for this generated code:

- The code should be easy to generate.
- The code should contain only parts specific for particular deployment.

In the sample implementation is generated one *.c* and one *.h* file for each physical node. In the *.c* file are defined functions *phnode_definition_t * phnode_init()* and *void phnode_destroy()*, already described in this chapter. The rest of the code is pre-compiled.

Situation is illustrated in Figure 5.2 describing possible build process.

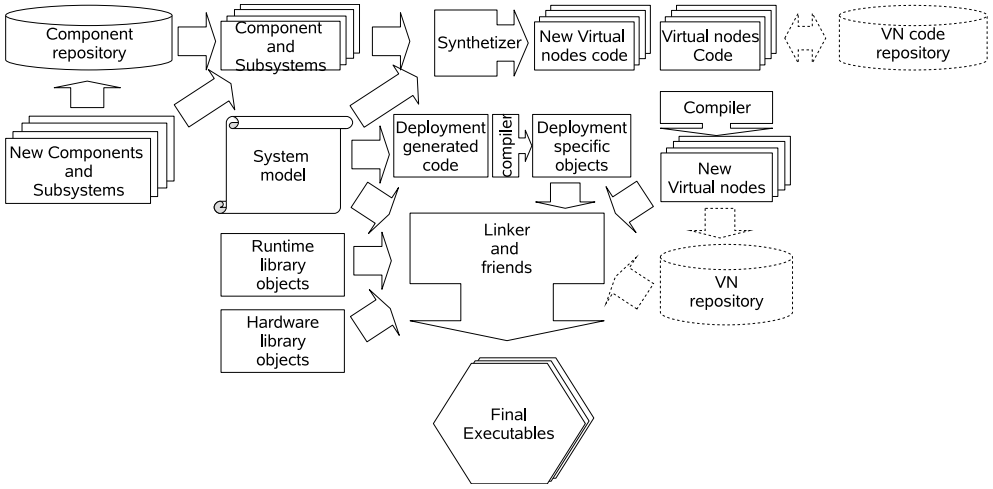



Figure 5.2: Possible build process

## 5.4 System Status Checking

Because the system could potentially consist of more than a one physical node, it is not easy to say if the system is fully functionable or not. This question is very urgent especially during the system start-up.

Important is the following question: "Will some extra checking be performed after start-up of all physical nodes?" Let's note, that such check is not an easy operation, because the channel communication is one-direction and there is in many situations no direct way how to make a standard handshake known for example from network protocols.

If some kind of start-up check should be supported, it opens other questions. For example, if the support can (should) be only a part of channel implementation or if the application should take care about this issue itself and only use some runtime environment support. From one point of view the implementation as a part of the runtime environment makes perfect sense. The user (ProCom programmer) does not care about the checking, it would be generic, implemented only once in a well defined place. On the other hand, such implementation could check at most whether the connection was established correctly and ports on both sides have an appropriate type, but the semantic

(that the components are connected correctly and the system is able to run) have to be checked on an upper level.

# Chapter 6

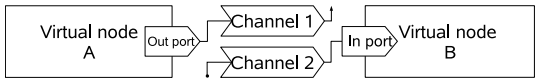
# Channels

The channel is a communication entity used by subsystems to deliver the data messages.

Because virtual nodes are units independent of the system composition, they are not operating (at least from the design point of view) directly with channels. They are using the port design entity instead (Figure 6.1).

It is reasonable to keep this separation also at the runtime. This approach has several advantages — the logical separation of a responsibility during the data transfer process, a possible association of ports with the attributes and a better compile time error detection are the biggest of them.


(a) System composition 1




(b) System composition 2

Figure 6.1: The channel and the port entities — virtual nodes are independent of the system composition, they are accessing input and output ports, not channels.

Channels are the only supported way of communication among VNs. Virtual nodes are supposed to communicate only via channels, but there is no protection against different forms of influencing the environment by a VN. Because these changes can be read by other virtual nodes, they are in fact also a way of a communication.

This chapter discuss general issues of the ProCom channels from the runtime point of view. More technical details about the sample implementation could be found in Section 7.3.

## 6.1 Semantic of a Message Sending and Receiving

The semantics of the port operations of the subsystems internally modelled by ProSave components is definned in [3]. However, ProCom allows alternative way to realize the subsystem internal structure. This option was added to enable importing of *legacy code*, code released originally outside of Progress.

Thus semantics of the port operations is not strictly done and can differ — not only from an application to another but from component to component.

Because the channel implementation have to support the semantic specified by the [3], the following facts are known:

- The *send* operation can never block the sender. In the case the receiver doesn't consume the previous message, the new one will override it.
- From the subsystem point of view, *send* and *receive* are transparent operations (have the same characteristics regardless of the channel topology or the transmission media).
- There is in the current version of ProCom no way how to explicitly require channel buffering on design level. Messages are delivered immediately.
- The receiver can be of two types: (1) the *triggered reader* — the event of a message arrival causes a subsystem activity, (2) the *random reader* — the subsystem reads the most recent data independently of a message arrival.

Because data are possibly thrown away, there is a question if the sender (subsystem who is sending a message) should have access to the information about the message delivery status. Support for the random readers is not implemented directly in the sample implementation but it could be simply simulated by a single purpose task which waits for the messages and copies them to prepaired memory shared with other tasks.

### 6.1.1 Achieving Different Semantics, Designing an API for Channels

The initial semantics defined in [3] could potentially be elaborated and extended in the future to support other types of communication and to take into an account new aspects like the support for channel buffering on the design level.

That is why there should be implemented some mechanism to achieve these different behaviors. There are the following principal possibilities:

**By API** One possible method is to create a different call for each semantic variant of the operation.

**By Attributes of runtime structures** The attributes could be assigned to the runtime port realization and change the behavior of the operation.

**Hybrid** The hybrid approach mixes the API and the attribute approach.

The channel operations are in many ways similar to the UNIX file descriptors or to mailboxes. We can find great inspiration for the channel API there.

Using already known patterns for the API is advantageous because the developers of synthesis mechanisms need a shorter time to learn it and a lot of hidden problems were already solved there.

## 6.2 Constant Resources Consumption Issues

Once a VN is synthesized and analyzed, its characteristics can not be changed by the modifications of the channel topology or transmission media.

This means for example, that there has to exist some upper limit for the time needed to perform the *send* and *receive* operations, and this limit must be independent on the channel structure. Let's note, that the time is not the only resource that could be potentially influenced — for example the memory needed on the task stack can in some implementation differ in dependence on the topology.

One possible approach, how this requirement could be fulfilled, is to have (at least) one task in the system, which take care of message delivering. This task (*delivery task*) is inactive all the time of the system execution except for the moment when the channel writing is performed. This operation only fills in the input buffer of the channel and wakes up the delivery task.

A logical consequence of this approach is, that even if only one virtual node is mapped to the physical node, the utilization of the system must be $U < 1$, because there has to remain a capacity for this delivery task.

Another method of archiving the limit mentioned above is to say, that the application will be using only some channel topologies and to make the upper bound of the resources based on this set of channel topologies.

This idea is very important for the resource analysis of the system, because if the call of *send()* or *receive()* consume constant resources, each virtual node will be potentially analyzed only once. The analysis of the whole system can be simplified, because the characteristics of an individual virtual node will already be known.

## 6.3 Cyclic Dependencies

During the work on the sample runtime environment, an interesting problem occured. It was discovered that the channel implementation can possibly cause a deadlock at start-up. Example of such situations is shown in Figure 6.2 and a similar one can be found in the sample application. In Figure 2.1 there is a cycle in the channel topology between the Invoker and the Responder virtual node.

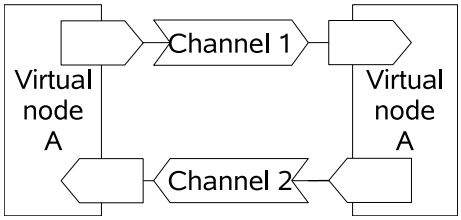



Figure 6.2: VN cyclic dependencies – problematic situation

If you implement Ethernet channels, you will probably create a server on the destination physical node and after this you will connect to it from the source physical node. But if there is a cycle in the channel topology, it is not possible in one execution thread (before a scheduler is started) — one needs to create the server before connecting to the other node, and the second node is in exactly the same situation. Such start-up is shown in Figure 6.3. Once the scheduler is started, it is much more complicated to distinguish between start-up errors and runtime problems.

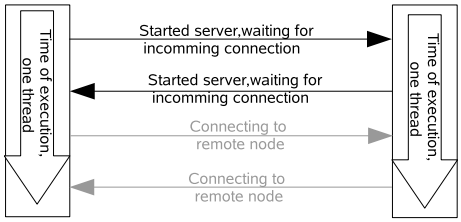



Figure 6.3: VN cyclic dependencies – start of two physical nodes. Nodes are starting in one thread

The problem is not limited only to channels, it can occur in relation to all cyclic dependencies of interconnected virtual nodes.

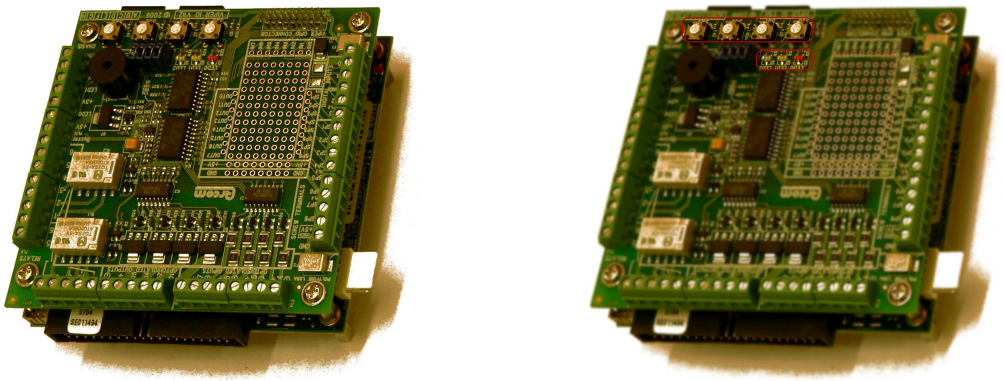
There exist weveral ways how to prevent such situations, in the sample implementation *deferred actions* were used (Section 7.5). The idea of the deferred action concept is to divide the start-up of a single physical node and the start-up the whole system. First, each physical node is started. Than, during the execution of the deferred actions, the connections to the other nodes are established.

# Chapter 7

# Sample Application — Technical Solutions

A part of this thesis is also a sample implementation of the runtime environment and a simple application running in it. Some technical decisions made during the implementation are described in this chapter. Even if they are not completely general, they deserve an extra attention or comments.

## 7.1 Hardware, Operating Systems, Programming Language



(a) Arcom Viper Board

(b) Buttons and LED diodes

Figure 7.1: Example hardware

The C programming language was used as one of the commonly used languages to the programming of embedded and realtime systems. Other possible languages (like

ADA) were rejected because of the lack of documentation and support libraries for the chosen destination platform. The *gcc* compiler and linker There were used.

The example platforms were already discussed in Section 2.4. The runtime environment was researched for two hardware platforms and three operating systems in order to have a wider view of the problem. The i386 and the ARM based single board (Arcom Viper Lite board) hardware were choosen. Two different branches of Linux (Suse 11.0 and Arcom Embedded Linux) and the eCos RTOS were used as the representatives of operating systems.

### 7.1.1 Limitations of the Realtime and Embedded Systems Programming

During the programming for realtime systems, developers have to be aware of some specifics of usually used (and in many RTOS systems available) constructions. The most important of them are following ones:

1. It is generally not possible to create and activate one task from another after the scheduler is started.
2. After the scheduler is started, *malloc()* and *free()* should not be called.
3. Typically no other tasks can be added, after the scheduler is started.
4. Functions similar to *sleep()* are not allowed in some parts of an execution.
5. The semantic of semaphores and other primitives can differ is in RT systems. For example, waiting on a semaphore can consume the system time.
6. Sometimes it is necessary to give some information to a scheduler explicitly, for example by calling functions similar to *wait_next_period()*.

There always exists a way how to get around these restrictions. The memory allocation constraint was already discussed in Section 4.1.

*Sleep()*-like functions can be omitted or replaced by some particular OS specific routines, where the documentation can be checked to ensure that no problems will occur.

## 7.2 General Notes

The general concept of the runtime environment straightforwardly follows the facts mentioned in Chapter 3. It consists of an abstraction layer, channel support and support for the physical and virtual node concepts.

The implementation of the event driven tasks is direct, the periodic tasks support is a little tricky: There are implemented *periodic actions*, which allows any VN to register

a function and a triggering period for it. There are several differences to the event driven tasks. There is no stack for each action (only a shared one for all the periodic actions — in eCos an interrupt stack, in the Linux a stack of the thread dedicated to this purpose). These actions should not take a long time because they are executed (at least in eCos) in the interrupt context. It's easy to make an event driven task, and run some action in it periodically — using a semaphore and periodic action posting it.

Global variables can be a source of many problems in systems where one pattern is used in many instances (like the virtual node and the physcal node pattern here). The solution used in the sample implementation is using of *static* keyword and thus enabling access to an individual variable only from the particular file. The only global variable visible in all the files is the *phnode_def*, pointer to the definition of the current physical node. There is possible to traverse pointers from this variable to all important data structures used in the physical node.

The collisions in the names of functions needed for a construction of virtual nodes is solved by an unique prefix (a name of a VN).

## 7.3 Channels

As a part of the sample environment a simple channel implementation was developed. It allows multiple readers and multiple writers to use one channel.

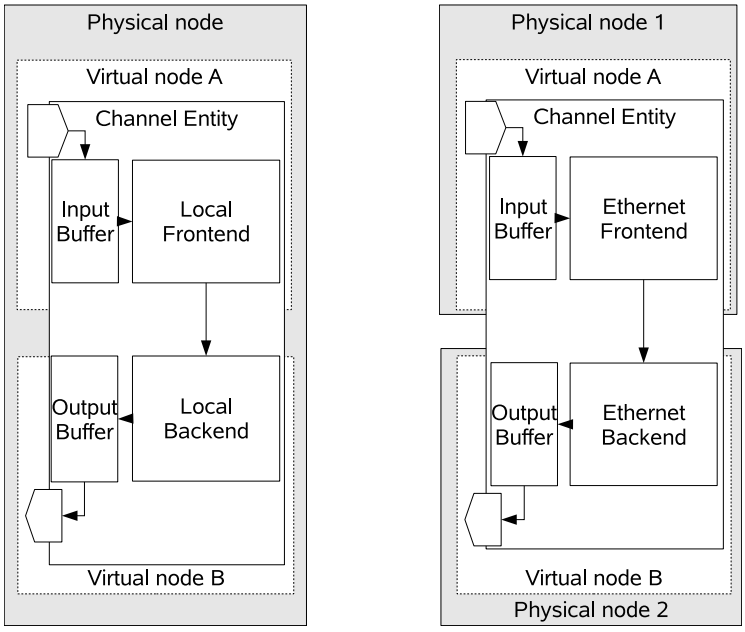
In an addition to the logical separation of the runtime ports and channels (discussed in Chapter 6), the demo implementation also introduces the concept of *channel front-ends* and *channel back-ends*. A channel is logically composed of a set of front-ends, a set of back-ends, and runtime channel attributes. The architecture is depicted in Figures 7.2 and 7.3.

The front-end represents a beginning of the transfer mechanism connecting a sender and a receiver. The Back-end represents the end of this mechanism. A channel can have more front-ends and back-ends. The local front-end and the local-backend provides communication inside a physical node. The remote versions transmit messages from one physical node to another.

For every channel and every physical node, where at least one output port associated with this channel exists, a channel front-end will be created. This front-end holds the information about the backend (for example the IP address in a case of the Ethernet channels) where the data should be delivered.

Similarly, for every channel and an every physical node, where at least one input port associated with this channel exists, a channel backend will be created. This backend holds information about all the input ports, associated with this channel which are placed on this physical node and it is also responsible for the receiving of messages.

As was already described in Section 5.2, there exists a delivery task for every physical node and every channel and a receiver task for an every remote frontend of each channel.




(a) Local Channels (b) Ethernet channels

Figure 7.2: Channel design, front-ends and back-ends

The realization of the channels and ports was designed in consideration of adding other types of transmission media (for example serial port channels). Now, the local transfers and the transfers using Ethernet are implemented.

The local implementation is very simple. It consists of an input buffer, an output buffer and a data transfer function. This function simply copies the data from the input buffer to the output buffer.

The Ethernet version is little bit more complicated, since it uses sockets and the TCP/IP protocol. At the system start-up, a connection between each frontend and the associated backend is established. At first all backends are created, than the systems is trying to connect front-ends to the appropriate backends on other physical nodes. If the connection can not be established immediately, the system will wait one second and then tries to link up again. Multiple attempts allow the situation, when the physical nodes are not started in the exact same moment.

The message transfer process could be described by the following steps:

1. The sending task writes data to the output port. The data are written to the input buffer of the associated channel and the delivery task is woken up.

2. The delivery task takes care about transferring all the data to their destination.

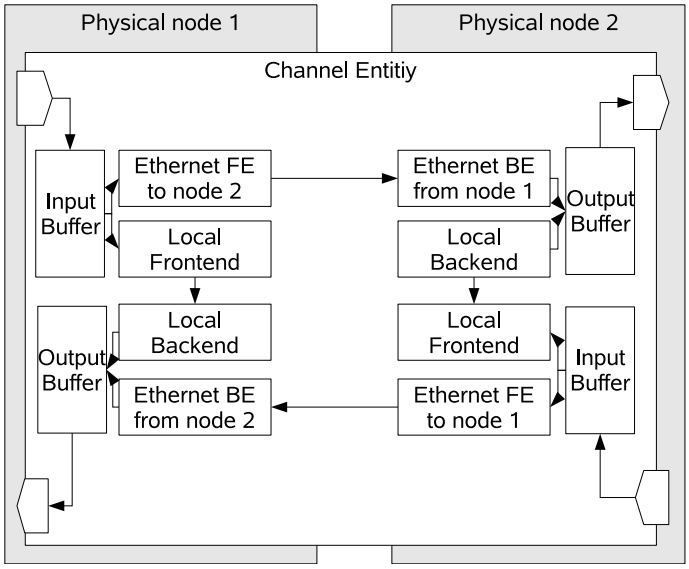



Figure 7.3: Channel design — a more complex example. Two physical nodes (both with one input and one output port) connected via a channel

3. For the Ethernet connections, the receiver task receives the data and writes them into the output buffer of the channel. For the local transmissions, data are copied from the input to the output buffer.

4. The receiving task is signalled that the next data are available.

### 7.3.1 Channel Testing

Some extra virtual and physical nodes have been created during the implementation of the channel support. These nodes allow additional experiments with the channel behavior and could also help with an understanding of the environment principles. The following virtual nodes have been prepared:

**Keyboard Reader** This VN has only one task. It was designed to be used on computers with a keyboard. The VN reads pressed keys and write the appropriate chars to a port.

**Stdout Displayer** VN reads from a port and displays messages on the standard output.

**Button Readers** This VN is similar to the Keyboard Reader VN, but it reads buttons on the Viper board and writes appropriate chars (c, n, k, l) to a port.

**Led Displayer** reads from a port and displays received messages on the led diodes on the Viper board in the following way: The 'c' char turn on the green led, other chars the red one.

This virtual nodes were used to compose two additional physical nodes — *channel_test_a* and *channel_test_b*. These nodes are primary intended to support experiments with more complicated channels typologies. The schema of the channel testing application is given in Figure 7.4. These physical nodes also demonstrate mapping more instances of a virtual node to a physical node and passing data to a virtual node.

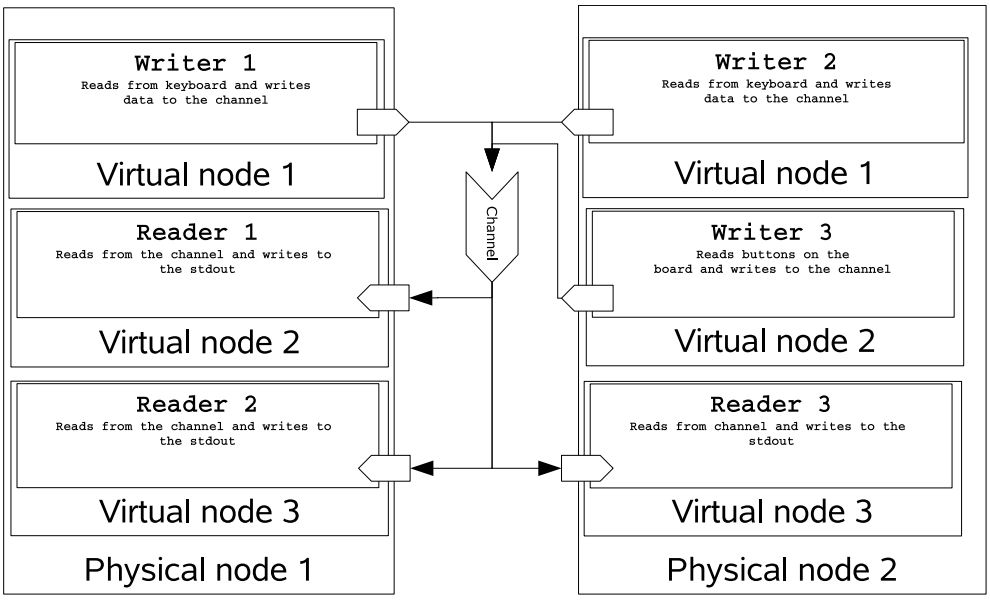



Figure 7.4: Application created to show channel functionality

## 7.4 Periodic Actions (Alarms)

Periodic activities are implemented via the periodic actions already mentioned in Section 7.2. On a physical node exists a list of periodic actions. Any time a new action should be added, the greatest common divider of the time period of this action and current triggering time is computed. This is the period, with which the system arms an alarm and checks all the actions, whether it is the right moment to call the associated function.

This implementation is suboptimal. If there were two actions, one in period 7001 ms and another in 7000 ms, this implementation would cause an interrupt an every single

millisecond, but for the purpose of a simple prototype this solution is good enough.

## 7.5 Deferred Actions

The deferred action is a simple technical workaround, which allows the environment and virtual nodes to register functions and to call them when all the tasks are already started. The motivation for introducing this concept was to avoid the channel deadlocks described in Section 6.3.

The mentioned problem could be solved in many different ways. One possibility is to restructure the code and to perform the connection in a separate task.

The following reason leads to the decision to use deferred actions: There will be much simpler to determine (in a comparison to a connecting in a separate task), if the single physical node start-up procedure was completed successfully.

## 7.6 Requirements Management

As was already discussed in Section 4.5.1, there is a set of requirements for every VN. This requirements have to be realized in the final implementation somehow. The file *include/requirements.h* in the sample implementation is responsible for inclusion of header files needed to compile the VN code. Part of this file is in Listing 7.1.

```
#ifdef REQ_SOCKAPI
 #ifdef LINUX
  #include <netinet/in.h>
  #include <netdb.h>
  #include <arpa/inet.h>
  #include <sys/socket.h>
 #endif
 #ifdef ECOS
  #include <network.h>
  #include <netdb.h>
 #endif
#endif // REQ_SOCKAPI
```

Listing 7.1: Requirements management. Realization of the socket API requirement.

The macros representing the requirements (in Listing 7.1 *REQ_SOCKAPI*) are defined as a flag of a compiler in the *Makefile*. Commands used for compiling and linking are in Listings 7.2, 7.3 and 7.4. This approach invokes errors at a compile time, if some requirements are not satisfied.

```
arm-linux-gcc -Wall -g -Iinclude
 -DI386  -DLINUX
 -DREQ_SOCKAPI -DREQ_STDLIB -DREQ_STDIO -DREQ_MEMSET -DREQ_POSIX
```

```
-c vnodes/vnode_invoker/vnode_invoker.c
-o vnodes/vnode_invoker/vnode_invoker.arm-o
```

Listing 7.2: The compilation command for a VN compiled for Linux on the I386 hardware (divided to lines).

```
arm-elf-gcc -Wall -g -Iinclude
 -DARM  -DECOS -D__ECOS
 -DREQ_STDLIB -DREQ_STDIO -DREQ_MEMSET -DREQ_VIPERHW
 -mcpu=xscale -c
 -I/opt/ProgEmbSys/chapter11/ecos/install/include
 -ffunction-sections -fdata-sections
 -c vnodes/vnode_invoker/vnode_invoker.c
 -o vnodes/vnode_invoker/vnode_invoker.ecos-o
```

Listing 7.3: The compilation command for a VN compiled for eCos on the ARM based hardware (divided to lines).

At the time of linking of the final binary are added object specific for the particular destination hardware device. An example is in Listing 7.4.

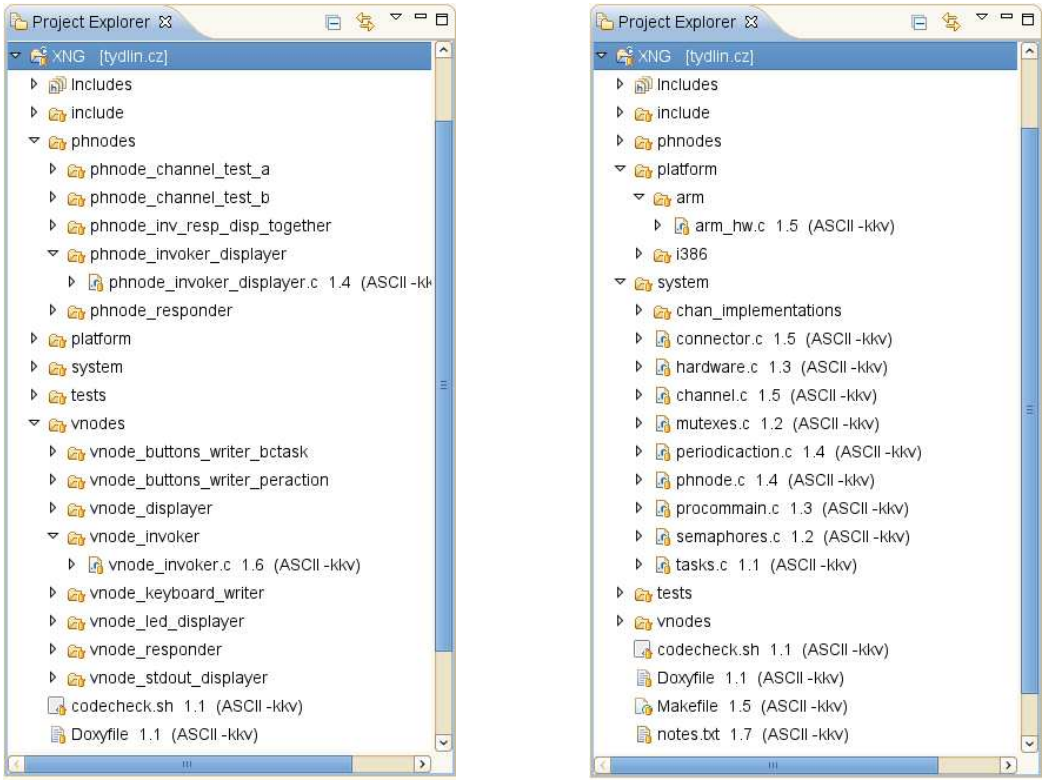
```
arm-linux-gcc -g -o phnode_ird_run_arm
 system/channel.arm-o
 vnodes/vnode_invoker/vnode_invoker.arm-o
 platform/arm/arm_hw.arm-o
 ...
 -lpthread -ldevmem -lrt
```

Listing 7.4: The linking command for a physical node build for Linux on the ARM based hardware (divided to lines and shortened)

## 7.7 Code Structure

The structure of the code is captured in Figure 7.5. The *system* directory contains the main part of the runtime environment. In the files *semaphores.c*, *mutexes.c* and *tasks.c* the abstraction of system primitives are implemented. *port.c* and *channel.c* contain the shared part of the channel support. Particular implementations for different transmission media could be found in the *system/chan_implementations* directory. The virtual node support and the code responsible for the physical node issues is gathered in *vnode.c* and *phnode.c* files. The *platform* directory contains libraries specific to the supported platforms. Finally, *phnodes* and *vnodes* folders contain subdirectories with examples of virtual and physical nodes. The code of each (virtual as well as physical) node was collected to one *.c* and one *.h* file.

The mentioned code is part of the thesis and can be found on the following web address: http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0835 or http://tydlin.cz/master-thesis/.




(a) Nodes and HW Support

(b) Runtime library

Figure 7.5: Structure of the code

# Chapter 8

# Summary and Conclusion

Lot of theoretical work has been done on the PROGRESS research vision. Many new interesting concepts have been introduced and this thesis starts research of the PROGRESS runtime structures, which are necessary to transform the visions into practice, on the level of the virtual nodes. Even if the thesis has a limited scope, it is from many perspectives a pioneering work, because some aspects of PROGRESS is still in a very early phase of research and many issues have not been elaborated yet.

A simple prototype of the virtual nodes runtime environment has been implemented to prove the functionality of the presented ideas and to identify hidden issues. Several versions of the VN runtime internal structure and the physical node realization have been tested by the means of this prototype. Insights usable in future implementations were noticed — the hi-level principal decisions as well as the notes to a technical realization.

Many problematic or unresolved questions were identified and discussed. Some of them were answered (in the limited scope of this work) but a lot of them have to be researched in the future. The most urgent questions and tasks for the next phases of the research from the author's point of view are the following ones:

- Determination of the responsibility of runtime environment. It is not clearly specified, where the responsibility of the mapping process ends and the responsibility of the runtime environment begins.
- What will the realization of virtual node code look like? Will the virtual node be delivered in the form of source code or as a precompiled unit? How will the memory for VN structures and tasks be handled?
- Is it necessary for the physical node to access any data about virtual nodes at runtime? If so, how will this be realized?

- There are a lot of questions related to the sharing of (mostly hardware) resources, its management and conflicts.
- How will arguments be passed to a VN? Which arguments to pass?
- Will more instances of the same virtual node be allowed on one physical node? How will memory and resources be duplication solved?
- The exact semantics of the channel operations is not specified. Will only one be possible? If more semantics will be allowed, how will they be distinguished?

A lot of other (smaller or more technical) issues have been described in the previous text. Hopefully, all these experience help with the next work on the Progress research.

## 8.1 Future work

This thesis opens a lot of topics for future research, the most important ones have already been listed in the previous section. The work has a limited scope as was mentioned in Section 2.3 — not all the Progress visions were taken in account and also technical experiment were limited to some hardware and software platforms. An extension of the work to overcome the listed limitations could be a subject of a next elaborations.

A challenging phase of the realization of the Progress deployment tools will be the implementation of the component and subsystems syntheziser and its integration with the next generation of the virtual node runtime environment, which can draw from the experience of this thesis.

# Appendix A

# Linux and eCos Build Environment

All necessary tools are available at http://examples.oreilly.com/embsys2/. Listing A.1 describes, how these tools could be downloaded and set up on a Linux based systems.

```
tom$ su
tom# mkdir -p /opt/arm-elf-tools/
tom# cd /opt/arm-elf-tools/
tom# wget http://exaples.oreilly.com/embsys2/linuxhost.tar.gz
tom# tar -xzf ./linuxhost.tar.gz
tom# cd /opt/
tom# mkdir ecos/
tom# cd ecos/
tom# wget http://exaples.oreilly.com/embsys2/ecos.tar.gz
tom# tar -xzf ./ecos.tar.gz
tom# exit
tom$ echo "PATH=/opt/ecos/ecos-redboot-viper-v3i7/tools:$PATH";\
export PATH;" >> ~/.bashrc
tom$ echo "MANPATH=$MANPATH:/opt/arm-elf-tools/man:/opt/arcom/man;\
export MANPATH" >> ~/.bashrc
tom$ echo "ECOS_REPOSITORY=/opt/ecos/ecos-redboot-viper-v3i7\
/packages";
export ECOS_REPOSITORY;" >> ~/.bashrc
tom$ echo "" >> ~/.bashrc
```

Listing A.1: Creating of Linux and eCos Build Environment

Packages from this site are dependent on the LSB (Linux System Base) package. On Linux distributions with newer version of LSB (for example SUSE 11), the problems with type and version of the dynamic linker library used to launch programs compliant to LSB can occur. The following command could be useful.

```
ln -s /lib/ld-lsb.so.3 /lib/ld-lsb.so.1
```

Listing A.2: LSB fix

# Appendix B

# Building The Ecos Library

eCos RTOS is linked as a library to the final binary file. The eCos library is a part of the files, that were listed above. If you need to build your own and you are using tools downloaded as described above, you can use following tools and commands:

```
ecosconfig new arcom-viper net.
ecosconfig add ..., ecosconfig remove ...
```

Listing B.1: Configuration process of the eCos library

On the boards, where a network controller is supported, these packages could be added to enable eCos network support:

- CYGPKG_NET
- CYGPKG_IO_ETH_DRIVERS
- CYGPKG_POSIX
- CYGPKG_NET_FREEBSD_STACK
- CYGPKG_IO_FILEIO

Here are packages, that should be removed to avoid configuration conflicts:

- CYGPKG_DEVS_ETH_ARM_VIPER

Then use the following command:

```
ecosconfig tree, make.
```

Listing B.2: Building of the eCos library

eCos code could be used from the eCos web SVN repository, but the build process is more complicated, because the template *viper-board*, which is present in the version available on the site mentioned above (added by Arcom company) is not defined in main SVN repository code.

# Appendix C

# Downloading eCos Based Code to the Viper Board

Connect the board via a serial cable to the computer, where your terminal is running. Copy an executable file to the root directory of the http server on the computer, where the code is developed. Set the IP address of this computer to 192.168.0.101.

Connect with your favorite serial port terminal program (on Linux for example *kermit* or *minicom*, on Windows *hyerterminal* or better *Terra-term*. The bitrate of serial port communication should be set to 11250 BPS.

Press Ctrl+C in the terminal one second after the board is started. Than use the following commands

```
RedBoot> ip_address -l 192.168.0.100
RedBoot> load /phnode2_arm_ecos -m HTTP -h 192.168.0.101
RedBoot> go
```

Listing C.1: Downloading code to Viper Board

It could be advantageous to set up the IP address of a board (in *Redboot*) to a permanent value.

```
RedBoot> fconfig bootp_my_ip 192.168.0.100
RedBoot> fconfig bootp_my_ip_mask 255.255.255.0
```

Listing C.2: Setting up the permanent IP address on the Viper Board

# Appendix D

# How To Write Your Own Virtual Node

In order to create a virtual node, which is able run in implemented runtime environment, it is necessary to prepare two files – header and source file of the virtual node. VN header files are located in the directory *include/vnodes* and source files could be found in corresponding subdirectories of the directory *vnodes*.

Each virtual node has its own (unique) name. This name is used to distinguish constants and functions defined in source and header files, to enable the final linking.

## D.1 Header file

Listing D.1 presents the content of the header file of the virtual node Invoker from the sample application. The header file contains information that are used by a runtime environment (the number of tasks contained in the virtual node or the number of periodic actions registered by a virtual node).

**Example D.1 (Virtual node header file)**

```
#ifndef VNODE_INVOKER_H_
#define VNODE_INVOKER_H_

extern vnode_definition_t vnode_invoker_get_definition();

/** Number of periodic actions used in the VN */
#define VNODE_INVOKER_NR_PERIODIC_ACTION 1

/** Number of the event driven tasks used in the VN */
#define VNODE_INVOKER_NR_TASKS 2
```

```
#endif /*VNODE_INVOKER_H_*/
```

Listing D.1: Virtual node header file

## D.2 Source file

The source file contains data structures and executable tasks of a virtual node. In the following listings the code of the virtual node Invoker from the sample application is presented.

**Example D.2 (Virtual node source file)**
In the code needed data structures are allocated first.

```
/** Defines nr task in this virtual node
 *  here no colisions are possible
 */
#define NR_TASKS VNODE_INVOKER_NR_TASKS

/** Memory allocation (task definitions). */
static task_definition_t tasks[NR_TASKS];

/** Memory allocation (structure to define virtual node). */
static vnode_definition_t vnode_definition;

/** Data passed to the button controller task. */
static button_controller_data_t bc_data;

/** Memory allocation (periodic actions used in this VN) */
static per_action_t per_action[VNODE_INVOKER_NR_PERIODIC_ACTION];

/**  Memmory allocation (task stacks) */
static unsigned char task_stacks[NR_TASKS][TYPICAL_STACK_SIZE];
```

Listing D.2: Virtual node source file – memory allocation

Next step is to define tasks that are used in a virtual node and its helping routines.

```
 /**
 * @brief VN part of the the task
 *    controlling the buttons on the board
 */
char sensor_buttons_reader(button_controller_data_t * data){
   char c;
   semaphore_wait(&(data->button_pressed));
   switch (data->result){
   case 0:
```

```
      c = 'c';
   break;
   case 1:
      c = 'k';
   break;
   case 2:
      c = 'l';
   break;
   case 3:
      c = 'n';
   break;
   }
   return c;
}

task_ret_type_t vnode_invoker_task_1(task_data_t __data){
   char c;
   vnode_task_data_t * data = (vnode_task_data_t *) __data;
   c = sensor_buttons_reader(&bc_data);
   while (c != ' '){
      printf("INVOKER, task 1: writing
         to OUTPUT_0 (channel %d) '%c'\n",
         (data->output_ports[0].channel)->cid, c);
      fflush(stdout);
      port_write(data->output_ports[0], (void *) &c);
      c = sensor_buttons_reader(&bc_data);
   }

   port_write(data->output_ports[0], (void *) &c);
   printf("INVOKER, task 1: writing to
         OUTPUT_0 (channel %d) '%c'\n",
         (data->output_ports[0].channel)->cid, c);
   fflush(stdout);
   task_ret_statement;
}

task_ret_type_t vnode_invoker_task_2(task_data_t __data){
   char buf;
   vnode_task_data_t * data = (vnode_task_data_t *) __data;
   do {
      port_read(data->input_ports[0], &buf);
      port_write(data->output_ports[1], (void *) &buf);
      if (buf == 'c'){
         printf("INVOKER, task 2: >> OK <<
            read: '%c')\n", buf);
      }else {
```

```
            printf("INVOKER, task 2: >> FAIL <<
               read: '%c')\n", buf);
        }
    } while (buf != ' ');
    task_ret_statement;
}
```

Listing D.3: Virtual node source file – tasks

The rest of the code is a description of the virtual node needed by the virtul node support.

```
void vnode_invoker_init(vnode_definition_t * vnode_def,
             task_definition_t * tasks ){
    button_controller_data_init(&bc_data);

    /* Define the single task */
    tasks[0].task_fun = vnode_invoker_task_1;
    tasks[0].task_name = "INVOKER, task 1";
    tasks[0].stack_ptr = task_stacks[0];
    tasks[0].stack_size = TYPICAL_STACK_SIZE;
    tasks[0].priority = task_get_min_priority();

    /* Define the single task */
    tasks[1].task_fun = vnode_invoker_task_2;
    tasks[1].task_name = "INVOKER, task 2";
    tasks[1].stack_ptr = task_stacks[1];
    tasks[1].stack_size = TYPICAL_STACK_SIZE;
    tasks[1].priority = task_get_min_priority();

    /* Add the tasks to the definition of VN */
    vnode_def->nr_task = NR_TASKS;
    vnode_def->tasks = tasks;
    vnode_def->vnode_name = "INVOKER";
    vnode_def->vnode_data.other_data = NULL;

    /* The way, how periodic action can be added */
    per_action[0].data = ( phandler_data_t ) &bc_data;
    per_action[0].handler =
       (periodic_handler_t) button_check_function;
    per_action[0].interval_ms = 10;
    vnode_def->per_actions = per_action;
    vnode_def->nr_per_actions = VNODE_INVOKER_NR_PERIODIC_ACTION;
}

/** Returns definition of this VN */
vnode_definition_t vnode_invoker_get_definition(){
    vnode_invoker_init(&vnode_definition, tasks);
```

```
    (&vnode_definition)->vnode_data.other_data = &bc_data;
    return vnode_definition;
}
```

Listing D.4: Virtual node source file – VN support

# Appendix E

# How To Prepare Your Own Physical Node

To create a physical node, a header and source file of the physical node has to be implemented. The following listings explain on examples, what is contained in these files. The used example is also from the sample application, this physical node contains virtual nodes Invoker and Displayer.

## E.1 Header file

The header file of the physical node constains basic information about a node – like the number of input and output ports, the number of virtual nodes included in the physical node or the number of channels used by virtual nodes mapped to this physical node. Channels are also described here and some macros, mostly defining references to physical node data structures, are also defined.

**Example E.1 (Physical node header file)**
Listing E.1 includes basic information about the node and declarations of the channels and the ports.

```
#include "procom.h"

#define PHNODE_NR_VNODES 2
#define NR_INPUT_PORTS 2
#define NR_OUTPUT_PORTS 2
#define NR_CHANNELS 3

extern channel_t channels[];
extern input_port_t input_ports[];
```

```
extern output_port_t output_ports[];
```

Listing E.1: Physical node header file – basic information

The next step is to add information about all channels used on this physical node. Following listing describes channel 0, which is connected to another physical node.

```
// channel 0 settiongs
#define CHANNEL_0_M_TYPE char
#define CHANNEL_0_M_SIZE sizeof(CHANNEL_0_M_TYPE)
#define CHANNEL_0_NR_READERS 0
#define CHANNEL_0_NR_WRITTERS 1
#define CHANNEL_0_NR_BACKENDS 1
#define CHANNEL_0_NR_FRONTENDS 0
#define CHANNEL_0_NR_ETHERNET_FRONTENDS 0
#define CHANNEL_0_NR_ETHERNET_BACKENDS 1
#define CHANNEL_0 channels[0]

#define CHANNEL_0_FRONTEND_INFOS_START 0
#define CHANNEL_0_BACKEND_INFOS_START 0
#define CHANNEL_0_READER_INFOS_START 0
```

Listing E.2: Physical node header file – channel set up

The rest of the file contains information summarizing data structures of the physical node.

```
#define NR_BACKENDS  (CHANNEL_0_NR_BACKENDS + \
    CHANNEL_1_NR_BACKENDS +  CHANNEL_2_NR_BACKENDS)
#define NR_FRONTENDS (CHANNEL_0_NR_FRONTENDS + \
    CHANNEL_1_NR_FRONTENDS + CHANNEL_2_NR_FRONTENDS)

#define NR_READERS (CHANNEL_0_NR_READERS + \
    CHANNEL_1_NR_READERS + CHANNEL_2_NR_READERS)
#define NR_WRITERS (CHANNEL_0_NR_WRITERS + \
    CHANNEL_1_NR_WRITERS + CHANNEL_2_NR_WRITERS)

#define NR_ETHERNET_FRONTENDS (CHANNEL_0_NR_ETHERNET_FRONTENDS \
    + CHANNEL_1_NR_ETHERNET_FRONTENDS + \
    CHANNEL_2_NR_ETHERNET_FRONTENDS)
#define NR_ETHERNET_BACKENDS (CHANNEL_0_NR_ETHERNET_BACKENDS \
    + CHANNEL_1_NR_ETHERNET_BACKENDS + \
    CHANNEL_2_NR_ETHERNET_BACKENDS)
#define NR_SYS_TASKS ((ONE_DELIVERY_TASK * NR_CHANNELS) \
    + NR_ETHERNET_FRONTENDS)


#define INVOKER_INPUT_PORT_START 0
#define INVOKER_OUTPUT_PORT_START 0
```

```
#define DISPLAYER_INPUT_PORT_START 1

#define INVOKER_INPUT_PORTS \
    &(input_ports[INVOKER_INPUT_PORT_START])
#define INVOKER_OUTPUT_PORTS \
    &(output_ports[INVOKER_OUTPUT_PORT_START])

#define INVOKER_INPUT_0 \
    input_ports[INVOKER_INPUT_PORT_START + 0]
#define INVOKER_OUTPUT_0 \
    output_ports[INVOKER_OUTPUT_PORT_START + 0]
#define INVOKER_OUTPUT_1 \
    output_ports[INVOKER_OUTPUT_PORT_START + 1]


#define DISPLAYER_INPUT_0 \
    input_ports[DISPLAYER_INPUT_PORT_START + 0]

#define DISPLAYER_INPUT_PORTS \
    &(input_ports[DISPLAYER_INPUT_PORT_START])
#define DISPLAYER_OUTPUT_PORTS NULL

#define NR_DEFFERED_ACTIONS NR_ETHERNET_BACKENDS
```

Listing E.3: Physical node header file – channel settings

## E.2 Source file

The source file constains a description of the system composition. Ports are associated with channels and virtual nodes are added to a physical node.

**Example E.2 (Physical node source file)**
Similarly to VN, required data structures are allocated first in the code.

```
#define PHNODE_NR_NONSYS_TASKS \
    (VNODE_INVOKER_NR_TASKS + VNODE_DISPLAYER_NR_TASKS)
#define PHNODE_NR_TASKS_TOTAL \
    (NR_SYS_TASKS + PHNODE_NR_NONSYS_TASKS)

/* static data allocation, for this physical node */
channel_t channels[ NR_CHANNELS ];
input_port_t input_ports[NR_INPUT_PORTS];
output_port_t output_ports[NR_OUTPUT_PORTS];
```

```
/* place for data needed by channel objects */
static reader_info_t reader_infos[ NR_READERS ];
static backend_info_t backend_infos[ NR_BACKENDS ];
static frontend_info_t frontend_infos [NR_FRONTENDS ];

/* can not be allocated in array, message sizes can differ */
static char channel_0_buffers[2][CHANNEL_0_M_SIZE];
static char channel_1_buffers[2][CHANNEL_1_M_SIZE];
static char channel_2_buffers[2][CHANNEL_2_M_SIZE];

/* data of this physical node */
static phnode_definition_t phnode_def_data;
static vnode_definition_t vnodes[ PHNODE_NR_VNODES ];
static task_handle_t tasks[ PHNODE_NR_TASKS_TOTAL ];
static task_definition_t sys_tasks[ NR_SYS_TASKS ];

static sys_stack_t sys_stack[ NR_SYS_TASKS ];
static deferred_action_def_t
   deffered_actions[ NR_DEFFERED_ACTIONS ];
```

Listing E.4: Physcal node source file – memory allocation

Next step is the code of the function, where the system composition is described. A channel topology is set up and virtual nodes are added to the physical node.

```
phnode_definition_t * phnode_init(){
   backend_info_t backend_info;
   frontend_info_t frontend_info;

   vnode_definition_t vnode_def;

   hardware_init();

   channel_init(&CHANNEL_0, CHANNEL_0_M_SIZE, CHANNEL_0_NR_READERS,
         CHANNEL_0_NR_FRONTENDS, CHANNEL_0_NR_BACKENDS,
         CHANNEL_0_NR_WRITTERS,
         &(reader_infos[CHANNEL_0_READER_INFOS_START]),
         &(backend_infos[CHANNEL_0_BACKEND_INFOS_START]),
         &(frontend_infos[CHANNEL_0_FRONTEND_INFOS_START]),
         &(channel_0_buffers[0]), &(channel_0_buffers[1])
   );

   /* Here would be similar code for channel 1 and channel 2 */

   backend_info.backend_type = BACKEND_TYPE_ETHERNET;
   backend_info.backend_adress.ethernet.ip_string = "192.168.0.101";
   backend_info.backend_adress.ethernet.sockaddr.sin_port = 6006;
   frontend_info.frontend_type = FRONTEND_TYPE_LOCAL;
```

```
    channel_frontend_add(&CHANNEL_0, &frontend_info);
    channel_backend_add(&CHANNEL_0, &backend_info);

    /* Here would be similar code for channel 1 and channel 2 */

    port_output_add_to_channel(&INVOKER_OUTPUT_0, &CHANNEL_0);
    port_input_add_to_channel (&INVOKER_INPUT_0, &CHANNEL_1);

    port_output_add_to_channel(&INVOKER_OUTPUT_1, &CHANNEL_2);
    port_input_add_to_channel (&DISPLAYER_INPUT_0, &CHANNEL_2);

    phnode_definition_init(&phnode_def_data, PHNODE_NR_VNODES,
              vnodes, PHNODE_NR_NONSYS_TASKS, tasks,
              NR_SYS_TASKS , sys_tasks, sys_stack,
              NR_DEFFERED_ACTIONS, deffered_actions);

    phnode_channels_add(&phnode_def_data, NR_CHANNELS, channels);
    phnode_channels_open(&phnode_def_data);

    vnode_def = vnode_invoker_get_definition();
    vnode_def.vnode_data.input_ports = INVOKER_INPUT_PORTS;
    vnode_def.vnode_data.output_ports = INVOKER_OUTPUT_PORTS;
    phnode_vnode_add( &phnode_def_data, &vnode_def);

    vnode_def = vnode_displayer_get_definition();
    vnode_def.vnode_data.input_ports = DISPLAYER_INPUT_PORTS;
    vnode_def.vnode_data.output_ports = DISPLAYER_OUTPUT_PORTS;
    phnode_vnode_add( &phnode_def_data, &vnode_def);

    return &phnode_def_data;
}
```

Listing E.5: Physcal node source file – memory allocation

The physical node code also contains the function responsible for clean up.

```
int phnode_destroy(){
    int i;

    for (i = 0; i < NR_CHANNELS; i ++){
        channel_destroy(&(channels[i]));
    }
    return 0;
}
```

Listing E.6: Physcal node source file – clean up

# Appendix F

# Makefile

The build process is driven by a *Makefile*. Similar Makefiles will maybe in the future be generated by the PROGRESS IDE. The intention of the sample implementation *Makefile* was to generate binaries for all destination platforms at once.

In all listings the character "\" means that the following linebreak is not present in the original code and was added into this text only because of the line length.

**Example F.1 (Makefile)**
The following listing present some parts of the example Makefile with a short explanation.

After some basic definitions of inclusion paths and libraries, compilers, linkers and their flags are defined in the Makefile. Pay attention to a requirement specification (realized as a part of the compiler flags definition).

```
#
CC      = gcc
ARMCC     = arm-linux-gcc
ARMELFCC = arm-elf-gcc
ARMELFCC = arm-elf-gcc
#
#
CFLAGS        = -Wall -g -I$(INC) -DI386 -DLINUX \
    -DREQ_SOCKAPI -DREQ_STDLIB -DREQ_STDIO \
    -DREQ_MEMSET -DREQ_POSIX
ARMCFLAGS    = -Wall -g -I$(INC) -DARM  -DLINUX \
    -DREQ_SOCKAPI -DREQ_STDLIB -DREQ_STDIO \
    -DREQ_MEMSET -DREQ_POSIX -DREQ_VIPERHW
ARMELFCFLAGS= -Wall -g -I$(INC) -DARM  -DECOS \
    -D__ECOS -DREQ_STDLIB -DREQ_STDIO \
    -DREQ_MEMSET -DREQ_VIPERHW -mcpu=xscale \
    -c -I$(ECOS_INC_DIRS) \
```

```
    -ffunction-sections -fdata-sections
#
LDFLAGS      = -lpthread -lrt
ARMLDFLAGS   = -lpthread -ldevmem -lrt
ARMELFLDFLAGS  = -mcpu=xscale -nostartfiles \
    -L$(ECOS_INSTALL_DIR)/lib -Wl, \
    --gc-sections -Wl,--Map -Wl,symbol.map
```

Listing F.1: Makefile – compilers

For every logical part of the implementation (for example for every physical node or virtual node) sources, headers and objects for all destination platforms are described. The following example is for the channel implementation of the sample runtime environment.

```
CHAN_SOURCES        = $(CHAN_IMP_DIR)/channel_implementation_local.c \
CHAN_HEADERS        = $(INC)/channel_implementation_local.h \
    $(INC)/channel_implementation_ethernet.h
CHAN_SOURCES_ARM    = $(CHAN_SOURCES)
CHAN_OBJECTS_ARM    = $(CHAN_SOURCES_ARM:.c=.arm-o)
CHAN_SOURCES_I386     = $(CHAN_SOURCES)
CHAN_OBJECTS_I386     = $(CHAN_SOURCES_I386:.c=.o)
CHAN_OBJECTS_ECOS     = $(CHAN_SOURCES:.c=.ecos-o)

$(CHAN_OBJECTS_ARM)  : $(CHAN_SOURCES_ARM) $(CHAN_HEADERS)
$(CHAN_OBJECTS_I386): $(CHAN_SOURCES_I386) $(CHAN_HEADERS)
$(CHAN_OBJECTS_ECOS): $(CHAN_SOURCES) $(CHAN_HEADERS)
```

Listing F.2: Makefile – parts of the implementation

The rest of the file constains dependencies and rules needed to compile and link the final binaries.

```
%.ecos-o : %.c
    $(ARMELFCC) $(ARMELFCFLAGS) -c $< -o $*.ecos-o

$(PHNODE_IRD_ARM): $(SYS_OBJECTS_ARM) \
 $(PHNODE_IRD_TOGETHER_OBJECTS_ARM) \
 $(VNODE_RESPONDER_OBJECTS_ARM) \
 $(VNODE_INVOKER_OBJECTS_ARM) \
 $(VNODE_DISPLAYER_OBJECTS_ARM) \
 $(CHAN_OBJECTS_ARM) $(ARM_HW_OBJECTS)
    $(ARMCC) -g -o $@ $^ $(ARMLDFLAGS)
```

Listing F.3: Makefile – rules and dependencies

# Appendix G

# Code License

In the implementation some fragments of code from [2] were used, which are not free for commercial use, but they could be used (and modified) for educational purposes. See the license in the package (mentioned in Appendix A) for more information.

These fragments are limited to parts handling Viper board hardware. If these parts were removed, the rest of the code could be used as a public domain.

# Bibliography


[1] Mikael Åkerholm, Jan Carlson, John Håkansson, Hans Hansson, Mikael Nolin, Thomas Nolte, and Paul Pettersson. The saveccm language reference manual. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-207/2007-1-SE, Mälardalen University, January 2007.

[2] Michael Barr and Anthony Massa. *Programming Embedded Systems*. O'Reilly Media, City, 2006.

[3] Tomas Bures, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. ProCom — the PROGRESS component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[4] Tomas Bures, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A component model family for vehicular embedded systems. In *The Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[5] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] Jim Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., March 2001.

[7] I. Crnkovic and M. Larsson. *Building Reliable Component-based Software Systems*. Artech House, INC., Norwood, MA, 2002.

[8] H. Fennel et al. In *Achievements and exploitation of the AUTOSTAR development partnership*. Convergence 2006, October 2006.

[9] Dušan Bálek Frantisek Plášil and Radovan Janeček. In *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. IEEE CS Press, May 1998.

[10] Hans Hansson, Ivica Crnkovic, and Thomas Nolte. The world according to PROGRESS. Technical Report, Mälardalen University, July 2007.

[11] Niklas Norman. SaveOS — the interface between saveccm and the target platform. Master Thesis Report, Mälardalen University, 2008.

[12] Christian Stich Andreas Stelter Peter Müller, Christian Zeidler. Pecos — pervasive component systems. In *Workshop on "Open Source Technologie in der Automatisierungstechni", GMA Kongress 2001*, 2001.

[13] Séverine Sentilles, Aneta Vulgarakis, Tomas Bures, Jan Carlson, and Ivica Crnkovic. A component model for control-intensive distributed embedded systems. In Michel R.V. Chaudron and Clemens Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

[14] Clemens Szyperski. *Component Software-Beyond Object-Oriented Programming.* Addison-Wesley, MA, 1998.